Adaptive Multidimensional Parallel Fault Simulation Framework on Heterogeneous System

Jingbo Hu, *Student Member, IEEE*, Guohao Dai, *Member, IEEE*, Liuzheng Wang, Liyang Lai, *Member, IEEE*, Yu Huang, *Senior Member, IEEE*, Huazhong Yang, *Fellow, IEEE*, and Yu Wang, *Fellow, IEEE*

***Abstract*—Fault simulation is a critical component of the automatic test pattern generation (ATPG) tool, which is widely used in chip development. The CPU–GPU heterogeneous system can accelerate fault simulation. However, existing work faces the following challenges: 1) *Path Divergence:* The simulation path of different faults is not uniform, which leads to low parallel efficiency of different GPU threads; 2) *Unbalanced Workload:* The load of different computing units is not balanced, leading to serious differences in the execution time of each part; and 3) *Poor Scalability:* When the circuit scale increases, the GPU memory is limited and the simulation has strong structural dependence, which makes the simulation difficult. In this work, we propose an adaptive multidimensional parallel fault simulation framework based on the CPU–GPU heterogeneous system. We adaptively select different simulation approaches according to different circuit scales. In detail, we use the fanout-free region (FFR) grouping method to solve the problem of path divergence. We also use a combination of static and dynamic load balancing to tradeoff data handling and the execution time of each computing unit. We limit the queue length used in the GPU to improve the scalability of the simulation. To further accelerate, we propose the 4-D parallel architecture on multiple GPUs. Extensive experimental results show that our fault simulator based on 8 GPU is 105.7× faster than the commercial tool on average. For tens of millions of gate-level circuits, our fault simulator based on one GPU is up to 25.9× faster than the CPU single-threaded simulator.**

***Index Terms*—Fault simulation, heterogeneous, multiple GPUs, very large-scale circuits.**

I. Introduction

FAULT simulation is a very important step in very large-scale integration (VLSI) design [2]. Given a digital circuit design and a set of test patterns [test vectors on the primary inputs (PIs)], fault simulation is used to evaluate the number of faults that can be detected using the test patterns. However, with the development of VLSI technology, the design complexity has increased exponentially [29], [30], [31]. For a common industrial chip with tens of millions of gates, fault simulation may take weeks or even months to complete [1]. Therefore, it is extremely important and urgent to explore ways to accelerate fault simulation.

In the past, a lot of work used CPU–GPU heterogeneous systems to accelerate fault simulation [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14]. However, they usually have three challenges, as shown in Fig. 1. The first challenge is that different GPU threads have different simulation paths, that is, path divergence, resulting in low parallel efficiency. In the GPU, program execution follows a single-instruction–multiple-thread (SIMIT) mode. When the fault simulation starts, different threads simulate different faults or patterns. The branch structures in the code make the calculations performed by the threads inconsistent, which affects the parallelism of the underlying GPU. The second challenge is that the workload on different computing units is not balanced. The computing unit can be the GPU card or the thread in the same GPU. When the computing units simulate different faults or adopt different patterns, the difficulty of fault detection changes significantly. As a result, the execution time of each computing unit is different, and the overall simulation speed is reduced. The third challenge is the poor scalability of the fault simulation algorithm. When the circuit scale increases, the data calculated by each thread in the GPU also increases. Due to the limited GPU memory, too much simulation data cannot be stored, which in turn leads to a reduction in the number of concurrent threads. On the other hand, fault simulation is very dependent on the data of the circuit structure, so simply splitting the circuit is not enough to complete the simulation.

In this work, we propose the adaptive fault simulation framework based on the CPU–GPU heterogeneous system to solve the above three challenges, as shown in Fig. 1. We first select different fault simulation algorithms adaptively according to the circuit scale. For small-scale circuits, we design a fault grouping scheme based on fanout-free region (FFR) to reduce the path divergence between different threads. In addition, we propose static and dynamic load balancing strategies related to different fault simulation stages. For large-scale circuits, we use the priority queue with a limited length on the GPU to improve scalability. We also extend the fault simulation to multiple GPUs and design the 4-D parallel fault simulation framework to adapt to different circuit scales. The main contributions of this work are as follows.

Manuscript received 26 January 2022; revised 3 June 2022 and 11 August 2022; accepted 14 September 2022. Date of publication 11 October 2022; date of current version 19 May 2023. This work was supported in part by the National Natural Science Foundation of China under Grant U19B2019, Grant U21B2031, Grant 61832007, and Grant 62104128; in part by the Tsinghua EE Xilinx AI Research Fund; and in part by the Beijing National Research Center for Information Science and Technology (BNRist). This article was recommended by Associate Editor C.-K. Cheng. *(Corresponding author: Yu Wang.)*

Jingbo Hu, Huazhong Yang, and Yu Wang are with the Department of Electrical Engineering, Tsinghua University, Beijing 100084, China (e-mail: yu-wang@mail.tsinghua.edu.cn).

Guohao Dai is with the Qing Yuan Research Institute, Shanghai Jiao Tong University, Shanghai 200240, China.

Liuzheng Wang and Yu Huang are with the Architecture and Design Department, HiSilicon Technologies Company Ltd., Shenzhen 518129, Guangdong, China.

Liyang Lai is with the Department of Electrical Engineering and the Guangdong Provincial Key Laboratory of Digital Signal and Image Processing, Shantou University, Shantou 515063, Guangdong, China.

Digital Object Identifier 10.1109/TCAD.2022.3213617

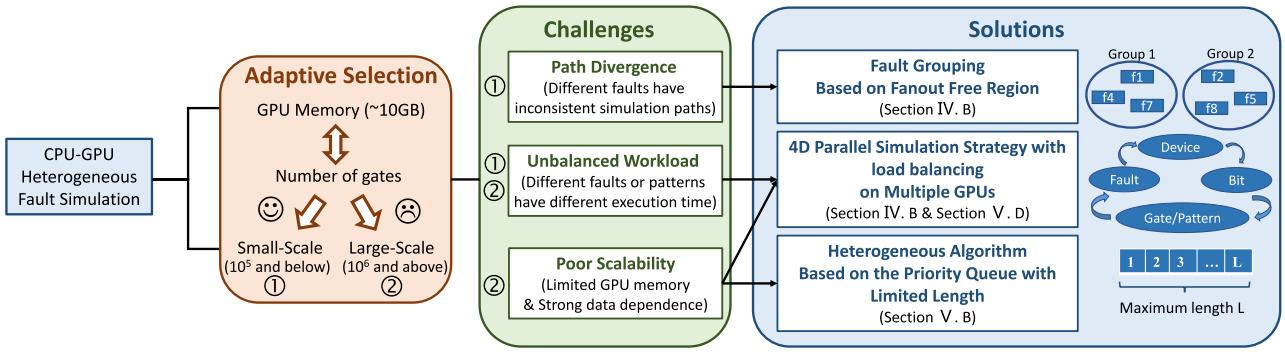

Fig. 1. Adaptive CPU–GPU heterogeneous fault simulation framework. We first judge the circuit scale based on the adaptation relationship between GPU memory and circuit information. For small-scale circuits, the biggest challenge is the low parallel efficiency caused by path divergence. For large-scale circuits, the biggest challenge is poor scalability. In addition, they also have the problem of an unbalanced workload. Therefore, we propose fault grouping, the 4-D parallel strategy with load balancing, and the heterogeneous algorithm that limits the queue length to solve the above problems.

1) We propose a fault grouping strategy based on FFR to reduce path divergence. The faults with the same FFR are divided into a fault subset (FSS). One subset can be simulated using the unified fanout region (FR). Therefore, the threads in the same warp on the GPU have the same simulation path, which improves the parallel efficiency.
2) We design the priority queue with a limited length on the GPU to improve scalability. Due to the limited GPU memory, we preset the maximum length of the queue used by each thread. The priority queue is to avoid invalid simulation. If the GPU cannot simulate the fault, that is, exceeding the queue length limit, it needs to be resimulated by the CPU.
3) We propose the 4-D parallel architecture with load balancing based on multiple GPUs. The four parallel dimensions are device-, fault-, bit-, and pattern/gate-parallelism. We also apply static load balancing on different GPUs to reduce data handling. Dynamic load balancing is applied between different thread blocks on the same GPU to improve the overall simulation speed.
4) Extensive experiments and analyses have been conducted on multiple circuits. The results show that our fault simulator for small-scale circuits based on a GPU is 22× faster than the commercial tool on average, and based on 8 GPUs is 105.7× on average. Our fault simulator for tens of millions of gate-level circuits based on a GPU is up to 25.9× faster than the CPU single-threaded simulator.

This article is organized as follows. In Section II, we introduce the GPU architecture and previous work to accelerate fault simulation with the GPU. Section III introduces the problem definition and initialization of fault simulation. Our approaches and optimization strategies for circuits of different scales are shown in Sections IV and V, respectively. The experimental results are reported in Section VI. Finally, Section VII concludes the work.

II. Background

A. GPU Architecture

We first describe the overall architecture of the general-purpose GPU. From a physical perspective, the GPU contains multiple streaming multiprocessors (SMs), and all SMs share global memory. Each SM contains multiple streaming processors (SPs). Multiple SPs in the same SM can execute the same instruction simultaneously and can access unified shared memory. From a logical perspective, the GPU contains multiple thread blocks, and each block contains multiple threads. Each thread needs to follow the SIMT processing architecture to execute simultaneously. Therefore, the GPU is very suitable for implementing parallel algorithms based on low data dependence. For example, in the fault simulation, since different patterns and faults are independent of each other, the simulation can be performed in parallel by taking advantage of the GPU multiple threads.

B. Related Work

The first GPU-based fault simulation acceleration work uses a lookup table (LUT) for gate evaluation [19]. It uses data parallelism, i.e., each thread evaluates a gate based on different patterns and faults. Although it can achieve an average speedup of 34× compared with single-threaded commercial tools, there are still limitations in transmission overhead and the number of evaluation gates. There is also work trying to map the PPSFP [21] technology to the GPU architecture [20]. However, the cost of event scheduling and the cost associated with conditional branches in the GPU may make such methods less profitable. Chatterjee et al. [22], [23] used model parallelism to develop GPU-accelerated logic simulators. The circuit is first partitioned to be allocated to different thread blocks. However, this algorithm will bring unnecessary evaluation overhead. In [24], a 3-D parallel fault simulator based on GPGPU is proposed. The three dimensions of parallelism are block level, fault level, and pattern level. However, there is no bit parallelism for multiple patterns. Compared with two other GPU-based fault simulators [19], [20], it achieves more than three times speedup. Tong et al. [33] described an OpenCL implementation of stuck-at-fault simulation on the GPU. Due to the limitation of GPU memory, it effectively maps the netlist and analog value data to the GPU and deploys gate-, fault-, and pattern-parallelism. However, each work item needs to simulate the entire netlist and store logical values. When the circuit scale increases, the netlist needs to be partitioned, which reduces efficiency. In addition, there is some simulation acceleration work based on other fault models. Interested readers can refer to [25] for details.

TABLE I
FEATURE COMPARISON OF FAULT SIMULATORS

	[19]	[20]	[24]	[33]	Ours
Fault Grouping	×	√	√	×	√
Circuit Scale (The number of gates)	10^4	10^6	10^5	10^5	10^7
Static Load Balancing	×	×	√	×	√
Dynamic Load Balancing	×	×	√	×	√
Multi-GPU support	×	×	×	×	√

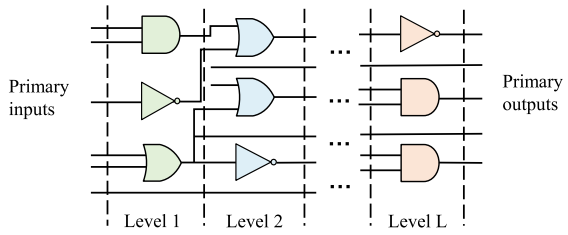

Fig. 2. Gates in the circuit except for the PIs are divided into 1–L levels according to the connection relationship. The PIs default to level 0.

To more clearly demonstrate the novelty and research contribution of our work, we compare the characteristics of our work with other fault simulators, as shown in Table I.

III. PRELIMINARIES

In this section, we give the problem definition and introduce the initialization process of fault simulation.

A. Problem Definition

The purpose of fault simulation is to detect the number of faults using a given series of patterns. The pattern refers to the value set V of *PIs*. The fault model we are based on is a binary stuck-at fault, that is, a fault is injected into the input/output line of a gate, with a value of 0 or 1. Before the fault simulation starts, a good machine simulation is required. On the premise that the circuit does not contain faults, the logic value of each gate is calculated according to the pattern V, denoted as t_{val}. Then, we inject the fault F into only one gate and calculate the fault value of each gate according to the pattern V, denoted as f_{val}. Finally, compare f_{val} and t_{val} on primary outputs (POs). If any one of the POs is different, it means that the pattern V can detect the fault F.

B. Initialization

When the CPU reads the netlist information, the circuit is levelized, as shown in Fig. 2. The level of each gate is one greater than the maximum level of its input gates. The level of PIs is 0. According to the above method, each gate is numbered in order of level. In addition, the CPU performs fault collapsing when counting the number of faults in the circuit. Then, we can get the type, fanin lists, and fanout lists of each gate. The CPU stores the above data as arrays, making it more conducive to the GPU execution and avoiding the use of structures that are not conducive to transmission. When we allocate memory, we call the `cudaMallocHost` function in CUDA to get pinned memory so that we can copy data to multiple GPU cards at the same time. In addition, we use compression encoding for partial arrays to reduce memory consumption and perform operations based on bit calculations.

IV. FAULT SIMULATION FOR SMALL-SCALE CIRCUITS

In this section, we introduce the details of our fault simulation approach for small-scale circuits. We first describe the overall workflow. Next, we propose the fault grouping approach and the multidimensional parallel fault simulation architecture (DFBP). Then, we introduce the two-level load balancing strategy. Finally, we analyzed the memory consumption.

A. Workflow for Small-Scale Circuits

The overall workflow of our proposed fault simulator for small-scale circuits is shown in Fig. 3. First, the CPU reads the netlist information and generates related data structures, such as fanins and fanouts. Then, n test patterns are randomly generated, and the CPU transmits data to GPUs. Next, GPUs perform good machine simulation. Each thread performs the logical calculation of the netlist for different patterns and stores the truth value array t_{val}. At the same time, the CPU groups faults and generates FRs of FSSs. After the above process, the CPU integrates the data transmitted back by GPUs and transmits the newly generated data to GPUs using a static load balancing scheme.

Then, the fault simulation starts. The workflow of each GPU card is the same, but for different FSSs. We use dynamic load balancing to make each thread block simulate different FSSs. The threads in one block use different patterns to simulate the same fault. When all faults are detected, each GPU transmits data to the CPU. After the CPU sums up, the final fault detection rate is obtained, and the whole simulation is ended. We implemented the proposed fault simulator based on CUDA [26].

B. Fault Simulation

1) Good Machine Simulation: We use device parallelism and pattern parallelism for good machine simulation. Different patterns are assigned to multiple GPUs, and each thread performs a parallel simulation of the complete circuit for different patterns to obtain the true logic value of all gates without fault, denoted as t_{val}. We also take advantage of the inherent bit parallelism on computer words to enable a thread to simulate 32 or 64 patterns at a time.

2) Fault Grouping Based on FFR: When faults are grouped, FFRs of the circuit need to be divided first, as shown in Fig. 4. The output of each FFR is a fanout stem (i.e., a gate with fanouts greater than 1 or the PO). Then, each FFR is divided into an FSS, and the FR of the stem represents the FR of the entire FSS.

3) 4-D Parallel Fault Simulation Architecture: We propose a fault simulation architecture for small-scale circuits called DFBP. DFBP is based on four dimensions of parallelism, as shown in Fig. 5. Each GPU receives the information of different FSSs and FR from the CPU and performs the simulation asynchronously through multiple streams in CUDA. For a GPU, multiple thread blocks simulate different FSSs. The threads in each block use different patterns (32 or 64 patterns per thread at a time) to simulate the same FSS in sequence. Each thread has an array f_{val} to store the fault value. On the other hand, we set the number of threads in a block to 32 to reduce the redundancy of pattern parallelism. Redundancy means that when one pattern has detected a fault, other patterns are still being tested in parallel. Therefore, we need to reduce the number of threads in the block as much as possible,

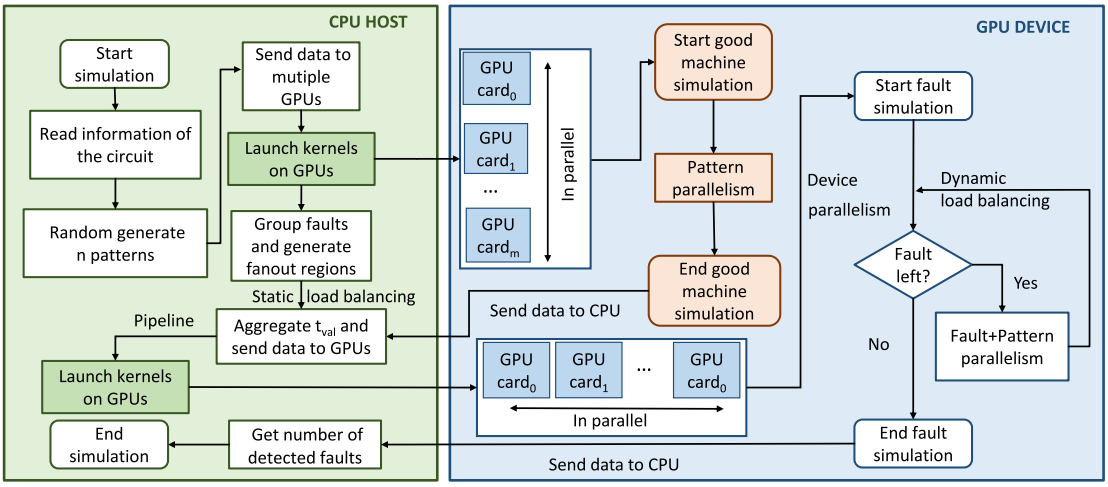

Fig. 3. Workflow of fault simulation on a heterogeneous platform with the CPU and multiple GPUs for small-scale circuits. The left side (mainly the green parts) is the process executed on the CPU. The right side is the process executed on multiple GPUs, including the main execution flow of good machine simulation (mainly the gray part) and fault simulation (mainly the yellow parts). And each GPU (the blue parts) is executed in parallel.

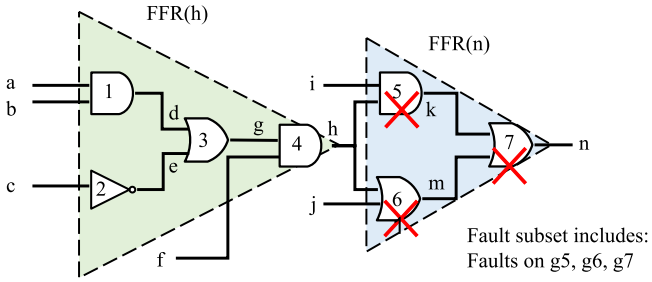

Fig. 4. Fault grouping based on FFR. The green region represents the FFR where the fanout stem is gate 4 (g4). The blue region represents the FFR where the fanout stem is g7. Gates in the same FFR are divided into an FSS.

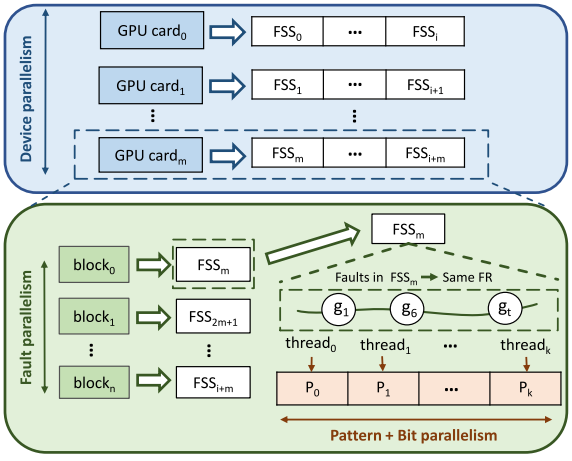

Fig. 5. 4-D parallel fault simulation architecture for small-scale circuits DFBP. DFBP includes device-, fault-, bit-, and pattern-parallelism. FSS: fault subset, FR: fanout region, and P: a set of 32 or 64 patterns.

while increasing the maximum number of physically reachable blocks. From the perspective of the GPU architecture, the underlying GPU calls warp to perform tasks, and a warp contains 32 threads. Therefore, we continue to reduce the number of threads in the block, resulting in reduced parallelism and waste of hardware resources.

The pseudocode of the fault simulation kernel for one FSS is provided in Algorithm 1. The first line determines the thread number in a block, which is represented by `threadIdx.x` in CUDA. Then, the fault is injected into the fanout stem of the FSS, that is, the truth value is reversed bit by bit. After each thread obtains the pattern index to be simulated (line 4 in Algorithm 1), it needs to traverse all the gates in the FR of the stem and perform logic calculations (lines 6–9 in Algorithm 1). When f_{val} and t_{val} on the PO are not the same, the pattern flag is set to 1 (lines 10–12 in Algorithm 1). Then, each fault in the FSS is detected in turn (lines 14–18 in Algorithm 1). Finally, after thread synchronization, we judge whether all faults in FSS have been detected (lines 20–22 in Algorithm 1). If not, use the remaining patterns to continue testing. Our method also performs fault dropping, that is, after thread synchronization, detected faults are flagged for deletion and are not executed in the next simulation.

C. Two-Level Load Balancing

We use the two-level load balancing strategy to further improve the performance of fault simulation. One level is static load balancing, which is used for device parallelism. The other level is dynamic load balancing, which is used for fault parallelism.

1) Static Load Balancing: When assigning FSSs to multiple GPUs, we adopt a static load balancing strategy. In our fault simulation algorithm (see Algorithm 1), there are two main parts, one is to traverse the FR, and the other is to detect faults in the FSS. Since the second part simply involves the logic calculation of a few gates, the time consumption of the first part is the main part. Therefore, we distribute FSSs to each GPU evenly according to the size of their FRs. Although some faults that are difficult to detect may have an impact on performance, we analyze that when the scale of faults is large, the above impact will be reduced. On the other hand, we avoid using more costly dynamic load balancing to increase the transmission overhead between the CPU and the GPU. The performance improvement is detailed in Section VI-C.

2) Dynamic Load Balancing: For a GPU, some FSSs are stored in its memory. Since the transmission overhead on the

Algorithm 1 Fault Simulation Kernel for an FSS

Input: true value t_{val}; fault value $*f_{val}$; fanout stem *stem*; number of patterns $p_$max; word size *ws*; array indicating whether the gate has been simulated (1 means simulated, while 0 means not) *flag*; netlist structure *net*; fault subset *FSS*;
Output: array indicating whether the fault has been detected (1 means detected, while 0 means not) *eval*;

```
1: tid = threadIdx.x;
2: fval[stem] =~ tval[stem];
3: FSS_d = 0;
4: for p_id = tid; (FSS_d! = 1)&&(p_id < p_max/ws); p_id = p_id + blockDim.x do
5:     p_flag = 0;
6:     for all gates i in FR of stem do
7:         //Compute the fault value of gate i
8:         compute(t_val, f_val, net[i], flag);
9:         flag[i] = 1;
10:        if i == PO then
11:            p_flag = p_flag|(f_val[i]^t_val[i]);
12:        end if
13:    end for
14:    for all faults j in FSS do
15:        //Compute the value on the stem based on fault j
16:        stem_val = feval(t_val, FSS[j], flag);
17:        eval[j] = (stem_val^tval[stem])&p_flag;
18:    end for
19:    __syncthreads();
20:    if detect all faults in FSS then
21:        FSS_d = 1;
22:    end if
23: end for
```

GPU is much smaller than the transmission overhead between the CPU and the GPU, we adopt a dynamic load balancing strategy for fault simulation between thread blocks. The specific scheme is as follows.

We use a global variable *fss_index* that indicates the index of the FSS. Each block obtains n FSSs and updates *fss_index* using atomic function (`atomicAdd`) provided by CUDA. When the simulation is completed, the block obtains the next n FSSs through *fss_index* and updates until all FSSs on the GPU are simulated. We adjust the size of n through experiments and find that the performance is best when $n = 1$. The performance improvement is detailed in Section VI-C.

D. Memory Analysis

According to the above fault simulation algorithm, the GPU memory is mainly occupied by the t_{val}, f_{val}, and FRs. They are all stored in global memory. The memory consumption of t_{val} is shown in (1), where N_p denotes the number of patterns, S_w denotes the word size in the computer (32 or 64), and N_g denotes the total number of gates in the circuit. v_t refers to the data type defined by the stored value

$$\text{Mem}(t_{\text{val}}) = N_p \div S_w \times N_g \times \text{sizeof}(v_t). \quad (1)$$

The memory consumption of f_{val} is shown in (2), where N_t denotes the total number of threads used by a GPU. N_t is equal to the product of the number of blocks (N_b) and the number of threads in each block ($N_{t\_in\_b}$). S_{FR} is the size of FR, and we select the maximum value of all FRs when we allocate the memory of f_{val}

$$\begin{aligned}\text{Mem}(f_{\text{val}}) &= N_t \times \max(S_{\text{FR}}) \times \text{sizeof}(v_t) \\ &= N_b \times N_{t\_in\_b} \times \max(S_{FR}) \times \text{sizeof}(v_t). \quad (2)\end{aligned}$$

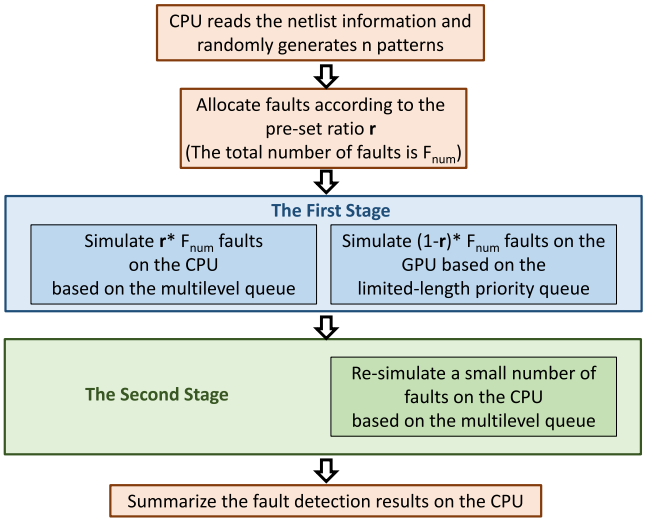

Fig. 6. Workflow of the two-stage fault simulation algorithm CGC.

Next, we take a circuit with ten million gates and 32 768 (32K) patterns as an example to analyze the memory when using one GPU.

1) For true logic value t_{val}, when the word size is 32, we calculate according to (1). When storing the value of a gate requires 4 bytes, we need 40 MB to store t_{val} for 32 patterns, and 40 GB to store t_{val} for 32K patterns.
2) For fault value f_{val}, when the maximum value of the FR is set to one million, we calculate according to (2). The total memory of storing f_{val} is $(4 * N_t)$ MB. When the total number of threads is 10 000, the occupied memory reaches 40 GB.

We take the GPU in our experiment as an example, and its memory is 8G. In order to enable the GPU to simulate circuits of tens of millions of gates and above, we propose the following three optimizations: 1) we transfer patterns to the GPU in batches to reduce the memory occupied by storing t_{val} in one simulation; 2) we limit the memory occupied by each thread to increase the total number of threads; and 3) the FRs of different stems are no longer used and stored. The details of our fault simulation algorithm for large-scale circuits are shown in Section V.

V. Fault Simulation for Large-Scale Circuits

The above method can effectively reduce the problem of path divergence and improve the parallel efficiency of the GPU. However, due to the need to store the FR of the fault gate on the GPU, the number of threads used is limited by the GPU's memory capacity. For very large-scale circuits, such as circuits with tens of millions of gates or more, the above-mentioned FR-based method may not be applicable. In view of this, we propose another algorithm, the two-stage fault simulation algorithm, called CGC. The overall workflow of the algorithm is shown in Fig. 6, and the pseudocode is shown in Algorithm 2. Next, we introduce CGC in detail.

A. Overall Algorithm Flow of CGC

After the CPU reads the netlist information and generates the corresponding data structure, the CPU and the GPU simultaneously start the fault simulation. To reduce the memory consumption of the GPU, we transmit 32 or 64 patterns at a time, depending on the word size (line 1 in Algorithm 2).

Algorithm 2 Two-Stage Fault Simulation Algorithm CGC

Input: true value t_{val}; fault value f_{val}; number of patterns $p_$max; word size *ws*; netlist structure *net*; number of levels in the circuit max_*dpi*; number of gates in one level **ngate_level*; ratio of faults allocated on the CPU *r*; num of faults *nof*; fault list *flist*; faults that the GPU cannot simulate *unf*

Output: number of faults finally detected *ndetect*

1: **for** $p\_id = 0$; $p\_id < p\_\max/ws$; p_id++ **do**
2: **for** $j = 1$; $j <$ max_*dpi*; $j++$ **do**
3: $num\_gate = ngate\_level[j]$;
4: //GPU uses the gate parallelism algorithm
5: //to perform good machine simulation
6: *GPU_good_sim*(num_gate, net, t_{val});
7: **end for**
8: //GPU simulation r*nof to nof fault
9: *GPU_fault_sim*($net, t_{val}, f_{val}, flist, r, nof$);
10: //CPU simulation 0 to (r*nof-1) fault
11: **for all** faults $f\_id < r * nof$ **do**
12: *CPU_fault_sim*($net, t_{val}, f_{val}, flist, f\_id$);
13: **end for**
14: $unf \leftarrow$ *memcpy_from_device_to_host*();
15: **for all** faults $f\_id < unf.size$ **do**
16: *CPU_fault_sim*($net, t_{val}, f_{val}, unf, f\_id$);
17: **end for**
18: **end for**
19: //The CPU merges the number of faults detected by itself and the GPU
20: $ndetect \leftarrow$ *CPU_merge_results*();

Algorithm 3 *GPU_fault_sim*($net, t_{val}, f_{val}, flist, r, nof$)

Input: true value t_{val}; fault value f_{val}; netlist structure *net*; ratio of faults allocated on the CPU *r*; num of faults *nof*; fault list *flist*; Maximum length of the priority queue *L*

Output: array indicating whether the fault has been detected (1 means detected, while 0 means not) *eval*;

1: $f\_s = threadIdx.x + r * nof$;
2: **for** $f\_id = f\_s$; $f\_id < nof$; $f\_id += nthreads$ **do**
3: $f = 0$;
4: $index = flist[f\_id]$;
5: $f_{val}[f] = compute\_val(index)$;
6: $fg[f] = index$;
7: $f++$;
8: **if** f_id is detected **then**
9: $eval[f\_id] = 1$;
10: **else**
11: *PriQ.push*(*out_gates*[*index*]);
12: **while** !*IsEmpty*(*PriQ*)&&$f < L$ **do**
13: $g = PriQ.pop()$;
14: $f_{val}[f] = compute\_val(net, g, t_{val}, f_{val}, fg)$;
15: $fg[f] = g$;
16: $f++$;
17: **if** f_id is detected **then**
18: $eval[f\_id] = 1$;
19: break;
20: **else**
21: //Push in the PriQ without repetition
22: *PriQ.push_new*(*out_gates*[*g*]);
23: **end if**
24: **end while**
25: **end if**
26: **end for**

Further, we use gate parallelism to complete good machine simulation (lines 4–7 in Algorithm 2). Each thread in the GPU calculates the values of different gates on the same level. Compared with pattern parallelism, gate parallelism is more scalable. That is, there is no linear relationship between the occupied memory and the total circuit scale.

Next, we assign faults to different hardware according to the ratio *r* of fault simulation. The GPU simulates the $r * nof$ to *nof* faults (line 9 in Algorithm 2). The CPU simulates the 0 to $r * nof$ faults (lines 11–13 in Algorithm 2), where *nof* is the total number of faults. To ensure the effectiveness of GPU simulation of very large-scale circuits, we limit the maximum length of the simulation path on each thread. A further problem is that certain faults cannot be simulated on the GPU. To reduce the proportion of such faults, the GPU should simulate short-path faults. Short simulation paths often correspond to small FRs. Therefore, we adopted a simple and effective method. When the gates are levelized, the faults on them can also be sorted according to the level. Intuitively, the faults in the front position may have a larger FR, that is, the number of gates on the simulation path may be more. Therefore, according to the simulation ratio, we assign the faults at the back to the GPU for simulation. Correspondingly, the remaining faults at the top are assigned to the CPU for simulation. The advantage of this method is that no additional complicated preprocessing is required. Another way is to sort the faults in descending order of FR. We still emulate the faults with the lower ranks on the GPU. Due to the need to count the FR of each gate, when facing large-scale circuits, the preprocessing time of this approach is significantly longer than that of the first one.

In the first stage, the CPU and GPU launch simulations in parallel. Although we optimize the fault distribution, it is inevitable that when facing large-scale circuits, there are still faults that the GPU cannot simulate. We call these faults super faults. In the second stage, all super faults are stored in a fault list and transmitted back to the CPU (line 14 in Algorithm 2). The CPU needs to resimulate all super faults (lines 15–17 in Algorithm 2). When all the simulations are completed, the CPU summarizes all the fault detection results (line 19 in Algorithm 2), including the faults detected by the GPU in the first stage and the faults detected by the CPU in the first and second stages. Finally, we get the fault detection rate.

B. GPU Fault Simulation Based on the Limited-Length Priority Queue

The pseudocode of the simulation on the GPU is shown in Algorithm 3. Each thread simulates different faults starting from $r * nof$ (line 1 in Algorithm 3). We first calculate the value of the fault gate. In order to further reduce the memory, we adopt the index structure, that is, we store the fault value in f_{val}, and store the corresponding gate number in *fg* (lines 3–7 in Algorithm 3). When the fault is not detected, we use the priority queue with a limited length [as shown in Fig. 7 (b)] to store the fanout of the fault gate, where the length is *L* (line 11 in Algorithm 3). Next, we traverse the gates in the priority queue in turn. When the fault value of the gate is different from its true value, we add its unvisited fanout gates to the queue. We repeat the above process until the fault is detected, the queue is empty or the maximum length of the queue is exceeded (lines 12–24 in Algorithm 3). Then, each thread starts the simulation for the next fault.

C. CPU Fault Simulation Based on the Multilevel Queue

The pseudocode of the simulation on the CPU is shown in Algorithm 4. Since memory is not a limiting factor on the

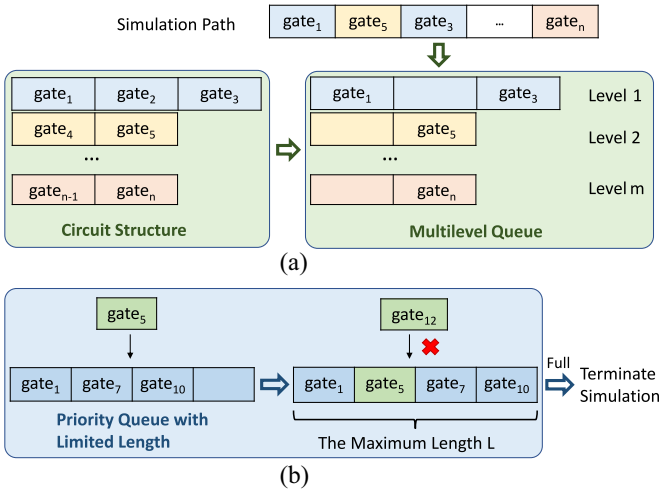

Fig. 7. Simulation data structures used on different hardware. The fault simulation on the CPU uses the multilevel queue. The gates in the simulation path are added to the corresponding positions in the queue according to the level. The gates in the unified level have no order. The fault simulation on the GPU uses the priority queue with a limited length. The gates in the simulation path are arranged in the queue from small to large. When the number of gates in the queue exceeds the maximum length of the queue, the GPU immediately terminates the simulation. (a) Simulation data structure used on the CPU. (b) Simulation data structure used on the GPU.

Algorithm 4 *CPU_fault_sim*$(net, t_{val}, f_{val}, flist, f\_id)$

Input: true value t_{val}; fault value f_{val}; netlist structure *net*; fault index f_id; fault list *flist*;

Output: array indicating whether the fault has been detected (1 means detected, while 0 means not) *eval*;

```
1: index = flist[f_id];
2: f_val[index] = compute_val(index);
3: if f_id is detected then
4:     eval[f_id] = 1;
5: else
6:     MlevelQ.push(out_gates[index]);
7:     for l = level[index] + 1; l < maxlevel; l++ do
8:         while !IsEmpty(MlevelQ[l]) do
9:             g = MlevelQ[l].pop();
10:            f_val[g] = compute_val(net, g, t_val, f_val);
11:            if f_id is detected then
12:                eval[f_id] = 1;
13:                break;
14:            else
15:                //Push in the MlevelQ without repetition
16:                MlevelQ.push_new(out_gates[g]);
17:            end if
18:        end while
19:    end for
20: end if
```

CPU, we can directly store the fault value in f_{val} without the index structure (lines 1 and 2 in Algorithm 4). When the fault is not detected, we use the multilevel queue for simulation (line 6 in Algorithm 4), as shown in Fig. 7(a). For a levelized circuit, we precalculate the number of gates on each level and allocate the corresponding space to form a multilevel queue. Then, gates whose logic values change in the simulation are added to the corresponding level queue. The advantage of the above-mentioned structure is that gates can be simulated sequentially, which prevents the occurrence of repeated calculations and further improves the simulation efficiency. We

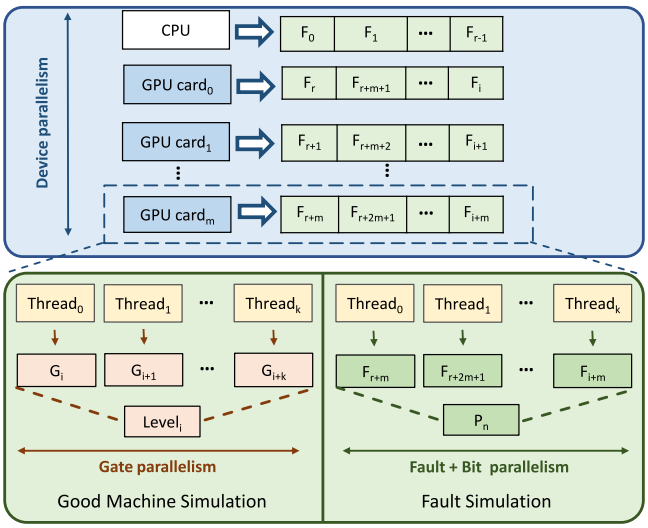

Fig. 8. 4-D parallel fault simulation architecture for large-scale circuits DFBG. DFBG includes device-, fault-, bit-, and gate-parallelism.

traverse each level after the level of the fault gate (line 7 in Algorithm 4). We simulate the gates in the queue in order. When their values change, we add their unvisited fanout gates to the queue. We repeat the above process until the queue is empty or the fault is detected (lines 8–18 in Algorithm 4). The CPU simulation in the first and second stages of CGC uses Algorithm 4.

D. 4-D Parallel Fault Simulation Architecture DFBG

For large-scale circuits, we can extend the fault simulation to multiple GPUs. We use parallel architecture DFBG (as shown in Fig. 8) to reduce memory consumption and improve computational efficiency. In the good machine simulation, different GPU threads simulate different gates on the same level, which is gate parallelism. In the first stage of fault simulation, the CPU and multiple GPUs simulate different faults in parallel, which is device parallel. In addition, in order to reduce the memory consumption of storing true values, different threads on the GPU adopt fault parallel simulation. Each thread simulates different faults for the same patterns, which is fault parallelism. Finally, each thread can simulate 32 or 64 patterns (depending on the word size in the computer) at the same time, which is bit parallelism.

E. Memory Analysis

Next, we analyze the memory consumption on a single GPU under the CGC algorithm. First, for t_{val}, since only 32 or 64 patterns are transmitted to the GPU in one simulation, the memory occupied by t_{val} is $4 * nog$ bytes, where *nog* is the number of gates in the circuit. In addition, for the storage of fault values, we preset its maximum length as L. The total memory required to store f_{val} is as follows:

$$\text{Mem}(f_{\text{val}}) = N_t \times L \times \text{sizeof}(v_t) \tag{3}$$

where N_t denotes the total number of threads used by the GPU, and v_t is the data structure used to define the logic value

$$\text{Mem}(fg) = N_t \times L \times \text{sizeof}(g_t). \tag{4}$$

In addition, we also need to store the gate numbers *fg* whose values change during the simulation (corresponding to f_{val}).

TABLE II
CIRCUIT INFORMATION

Circuit	Gates	Total Collapsed Faults	Fault Subsets	PIs	POs	Max Logic Depth
b17	30777	76625	9657	1452	1512	92
b17_1	38116	88098	10279	1452	1512	51
b18	111241	264091	34387	3357	3343	164
b19	224624	533276	69716	6666	6672	168
b20	19682	45459	5157	522	512	67
b21	20027	46154	5125	522	512	68
b22	29162	67536	7633	767	757	68
b18*	10,263,465	23,267,160	–	50355	50355	673
b19*	12,369,888	28,315,296	–	120096	120096	460
netcard*	13,365,616	17,721,088	–	1566376	782920	72
leon2*	13,351,445	16,750,210	–	1494445	747960	100
designA	10,082,130	29,481,882	–	998940	745420	164
designB	10,962,540	31,525,022	–	764760	560020	143

The memory consumption of fg is shown in (4), among them. g_t is the data structure used to define the gate number. In summary, we still take the circuit of ten million gates as an example. We set the total number of threads to 10 000 and L to 10 000. Then, the memory of t_{val} is 40 MB, and the memory of f_{val} and fg is 400 MB. Then, the total memory consumed by the above three parts is 840M, which is much smaller than the memory usage analyzed in Section IV-D.

VI. EXPERIMENTAL RESULTS

A. Setup

We evaluate the performance of our fault simulator on a set of large circuits from the ITC'99 benchmarks [27], IWLS 2005 benchmarks [32], and Industrial circuits (including designA and designB), as shown in Table II. Among them, the datasets with * are expanded datasets. The expansion method is as follows: b18* is obtained by splicing six copies of b18 longitudinally (depth) and then copying 15 pieces horizontally (breadth). b19* is obtained by splicing three copies of b19 longitudinally (depth), and then copying 18 copies horizontally (breadth). netcard* is obtained by copying eight copies of netcard horizontally (breadth). leon2* is obtained by copying five copies of leon2 horizontally (breadth). Since tens of millions of gate-level circuits (i.e., b18*, b19*, netcard*, and leon2*) use the CGC algorithm, we do not count their FSSs.

Experiments are performed on a Ubuntu 18.04.2 LTS server. The server has an Intel Xeon 2.2-GHz CPU E5-2630 v4 with 512-GB memory and 8 GPU devices. Each GPU device is a GeForce RTX 2080 Ti graphics card containing 68 multiprocessors and 4352 CUDA cores with a clock speed of 1.54 GHz and is equipped with 8 GB of available device memory. The NVIDIA CUDA 10.0 SDK toolkit is used for all the program execution. We use randomly generated patterns for fault simulation. In addition, our simulator uses 32-bit parallelism so that each thread can simulate 32 patterns at the same time. For each experiment, we perform 100 times to get the average result. In order to evaluate the correctness of our fault simulator, we use the open-source fault simulator FSIM [28] to verify the results. FSIM is a sequential simulator running on the CPU.

In the parameter settings of the following experiments, we grid-search the optimal parameters. When exploring the influence of a single parameter on the overall performance, the remaining parameters are set as optimal parameters unless otherwise specified.

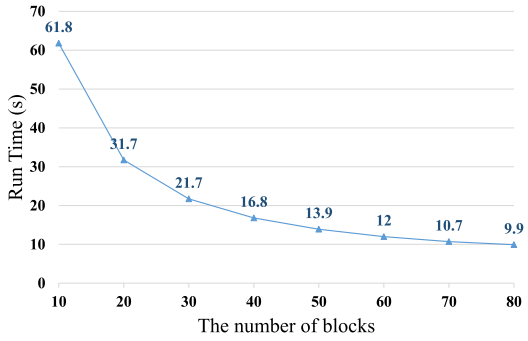

Fig. 9. Execution time of our fault simulator with different number of blocks (circuit b19, single GPU, and the number of threads in a block is fixed at 32).

B. Influence of Parallelism Factors

Considering the case of one GPU, two parallelism factors affect its performance. One is the number of thread blocks in the GPU, which is aimed at fault parallelism. The other is the number of threads in a block, which is aimed at pattern parallelism. We first explore the influence of the number of blocks on the performance of fault simulation. We use circuit b19 and fix the number of threads in a block to 32. The result is shown in Fig. 9. The execution time includes the time of initialization, data transmission, good machine simulation kernel, and fault simulation kernel. We can find that when the number of blocks keeps increasing, the execution time of fault simulation keeps dropping. The reason is that the number of FSSs of parallel simulation has increased, and the total number of threads used in the GPU, which is equal to the number of blocks multiplied by the number of threads in a block, has also increased, thereby improving the execution efficiency of GPU. However, it is worth noting that when the number of blocks increases to a certain range, the execution time tends to be flat. This is due to the limited number of physical cores in the GPU, and the performance is limited by memory usage.

On the other hand, we explore the effect of adjusting the number of blocks and threads on the performance of fault simulation under the premise that the total number of threads is equal. The results are shown in Fig. 10. Observing different circuits, it can be found that the performance is best when the number of threads is 32. Since the number of threads corresponds to the number of patterns simulated concurrently, when the number of threads increases, more patterns are executed for the same fault. When a fault is detected in one of the patterns, the other patterns will do repetitive total work. Therefore, the above process is likely to cause redundancy

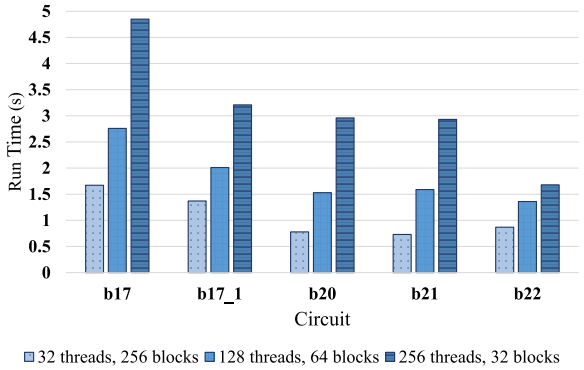

Fig. 10. Execution time of our fault simulator with different numbers of blocks and threads (single GPU). The total number of threads is set to 8192.

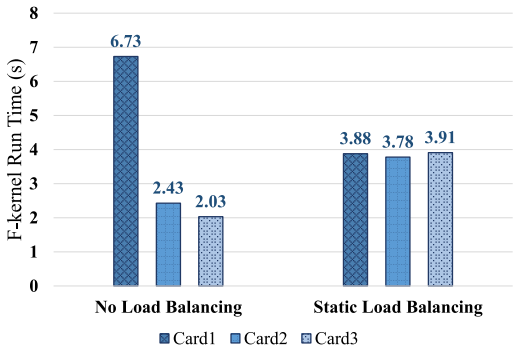

Fig. 11. Different GPU card's execution time of the fault simulation kernel with/without static load balancing (circuit b19 and three GPU cards). Each GPU card applies dynamic load balancing at the fault parallel level.

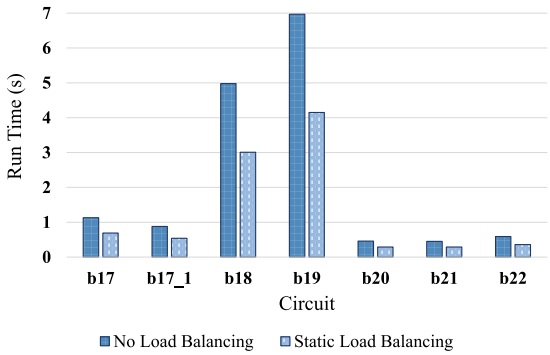

Fig. 12. Execution time of our fault simulator with/without static load balancing (three GPU cards).

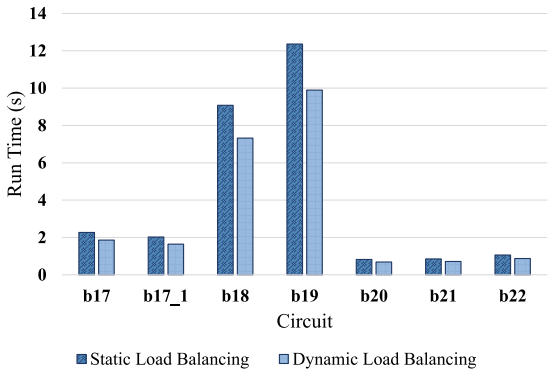

Fig. 13. Execution time of our fault simulator with different ways of load balancing (single GPU).

in simulation. In addition, each multiprocessor needs to call warps to perform operations in CUDA. Each warp contains 32 threads, so when the number of threads in a block is less than 32, it will cause performance degradation.

In summary, we adjust the above parallelism factors to the optimal situation in the following experiments, that is, the number of threads is 32, and the number of blocks is the maximum value that the memory can support.

C. Impact of Two-Level Load Balancing

1) Static Load Balancing: We apply static load balancing at the device parallelism level for multiple GPUs. We take three GPU cards as an example and perform a comparison with and without load balancing on circuit b19. The results are shown in Fig. 11. The execution time in Fig. 11 only refers to the time of the fault simulation kernel. We assume that there are $3 * m$ FSSs. When load balancing is not used, the FSSs are sequentially divided into three parts (1 to m, $(m+1)$ to $2*m$, and $(2*m+1)$ to $3*m$). Each part is assigned to a different GPU card. However, since the FRs in the first part of FSSs are significantly larger than the latter two parts, the simulation time of the three cards has a large difference. When using static load balancing, the size of the FRs in different parts is similar, so the execution time of each card is also similar. The effect of static load balancing on the total time of each circuit is shown in Fig. 12. The results show that compared with no load balancing, the overall performance of static load balancing has increased by 60%–70%.

2) Dynamic Load Balancing: We apply dynamic load balancing at the fault parallel level, that is, for a GPU, FSSs are dynamically allocated to each block. The result is shown in Fig. 13. Static load balancing is based on the size of FR in each FSS and FSSs are evenly distributed to each block. Compared with the preallocated static load balancing, dynamic load balancing has an average performance improvement of 23%.

D. Impact of Fault Grouping

We explore the impact of FFR-based fault grouping on the fault simulation performance. The results are shown in Fig. 14. The method without fault grouping means that each gate has a separate FR, instead of the gates in one FFR sharing the same FR. In addition, the faults on different gates need to be simulated separately, that is, each thread uses different patterns to traverse the FR of the fault and calculate the logical value. Finally, we detect whether the fault value can be transmitted to any one of POs. The results in Fig. 14 show that fault grouping can bring an average 2.4× performance improvement.

E. Performance Evaluation for Small-Scale Circuits

We first evaluate the impact of the number of GPUs on the performance of the fault simulation kernel. The results are shown in Fig. 15. It is worth noting that the vertical axis in Fig. 15 uses a logarithmic scale. Through the analysis of the results on different circuits, it can be seen that when the number of GPUs increases, the execution time of the kernel decreases. In the implementation of our fault simulator, we use the stream in CUDA to allow each GPU to perform simulation asynchronously, but there is still an overhead when creating, calling, and executing streams. When the number of GPUs increases, the proportion of the above overhead will increase.

TABLE III
PERFORMANCE ANALYSIS OF OUR FAULT SIMULATOR UNDER DIFFERENT NUMBERS OF GPUS

Circuit	Total Execution Time (in seconds) of Our Fault Simulator												Speed-up (x)			
	C.tool			nGFSIM [34]	Ours								C.tool			nGFSIM [34]
	1-T	*4-T*	*16-T*	*1-G*	*1-G*	*2-G*	*3-G*	*4-G*	*5-G*	*6-G*	*7-G*	*8-G*	*1-T*	*4-T*	*16-T*	*1-G*
b17	44.13	15.76	7.54	2.66	1.74	0.93	0.69	0.56	0.47	0.41	0.38	0.35	25.4	9.1	4.3	1.5
b17_1	41.09	14.71	7.92	2.43	1.43	0.75	0.54	0.43	0.37	0.33	0.30	0.28	28.7	10.3	5.5	1.7
b18	191.31	54.57	23.33	15.51	7.32	4.02	2.81	2.22	1.85	1.63	1.47	1.40	26.1	7.5	3.2	2.1
b19	408.93	110.27	45.34	21.86	9.89	5.36	3.90	3.17	2.75	2.47	2.24	2.10	41.3	11.1	4.6	2.2
Average Speed-up (×)													30.4	9.5	4.4	1.9

C.tool represents a commercial tool. n-T: the number of CPU threads is n.

TABLE IV
PERFORMANCE COMPARISON OF DIFFERENT FAULT SIMULATORS ON THE SAME PLATFORM

Circuit	Total Execution Time (in seconds)					Speed-up (×)				
						Ours (1-GPU)			Ours (8-GPUs)	
	FSIM [28]	*C.tool-t1*	*GPU-nG*	*Ours (1-GPU)*	*Ours (8-GPUs)*	*FSIM* [28]	*C.tool-t1*	*GPU-nG*	*FSIM* [28]	*C.tool-t1*
b17	23.92	44.13	2.63	1.74	0.35	13.7	25.4	1.5	68.3	126.1
b17_1	26.22	41.09	2.28	1.43	0.28	18.3	28.7	1.6	93.6	146.8
b18	155.33	191.31	11.17	7.32	1.40	21.2	26.1	1.5	110.0	136.7
b19	351.56	408.93	23.42	9.89	2.10	35.5	41.3	2.4	167.4	194.7
b20	6.83	7.05	1.21	0.68	0.18	10.0	10.4	1.8	37.9	39.2
b21	6.98	7.66	1.42	0.71	0.17	9.8	10.8	2.0	41.1	45.1
b22	9.66	9.73	1.93	0.87	0.19	11.1	11.2	2.2	50.8	51.2
Average Speed-up (×)						17.1	22.0	1.9	81.4	105.7

C.tool-t1 represents a commercial tool, running on the single CPU thread.

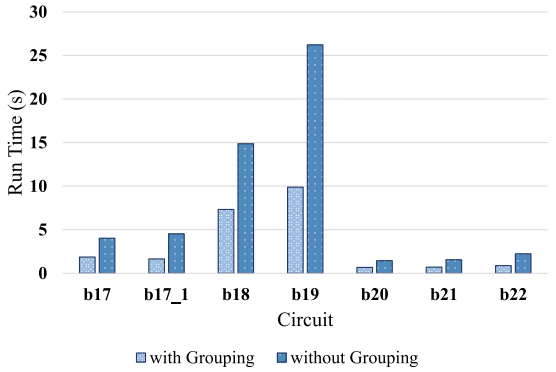

Fig. 14. Execution time of our fault simulator with/without fault grouping (single GPU). The GPU card applies dynamic load balancing at the fault parallel level.

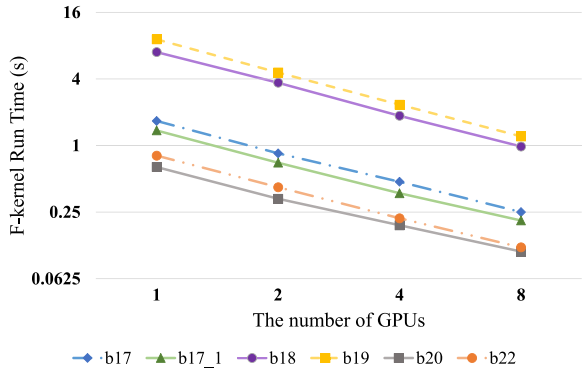

Fig. 15. Execution time of the fault simulation kernel under different numbers of GPUs. The vertical axis uses a logarithmic scale.

Therefore, the execution time of the fault simulation kernel has a nonlinear relationship with the number of GPUs.

Under the different number of GPUs, the total execution time of fault simulation for multiple circuits is listed in Table III. In addition to the execution time of the fault simulation kernel, the total time includes the time for initialization, data transmission, and execution of good machine simulation. It can be seen from Table III that when the number of GPUs continues to increase, the decline in the total execution time of the fault simulation gradually slows down. In other words, in addition to the inherent initialization time, the proportion of data transmission time increases, which reduces the overall performance. Therefore, when the workload on each GPU decreases, the data transmission time may become a performance bottleneck. Nevertheless, experimental results on four datasets show that our fault simulator is accelerated by an average of 1.9× compared to nGFSIM [34]. Compared to the commercial tool running under different CPU thread counts, it can speed up 4.4× to 30.4×.

Table IV shows the performance comparison results of different fault simulators. Among them, FSIM [28] is an open-source CPU fault simulator. GPU-nG is a single GPU fault simulator and no fault grouping is added. GPU-nG still includes three dimensions of parallelism: fault-, pattern-, and bit-parallelism. When only one GPU is used, compared with FSIM, our fault simulator can accelerate 17.1× on average. In addition, compared with the commercial tool, our fault simulator can accelerate by an average of 22×. Compared with GPU-nG, our fault simulator can accelerate 1.9× on average. When using 8 GPUs, our fault simulator is 81.4× faster than FSIM on average. In addition, compared to the commercial tool, the average speedup is 105.7×. For circuit b19, the acceleration is as high as 194.7×. In the case of unlimited memory, the performance is expected to be further improved for larger circuits, which shows the effectiveness of our fault simulator.

F. Performance Evaluation for Large-Scale Circuits

1) Simulation Results Under Different Methods: We first compare the simulation time under different methods (single CPU and single GPU), and the results are shown in Fig. 16. Among them, CPU-S is the CPU single-threaded simulation. CPU-M is the CPU multithreaded simulation, and the number of threads selected in the experiment is 32. CGC is our two-stage fault simulation method (see Section IV for details), and the experiment shows the results under the optimal fault

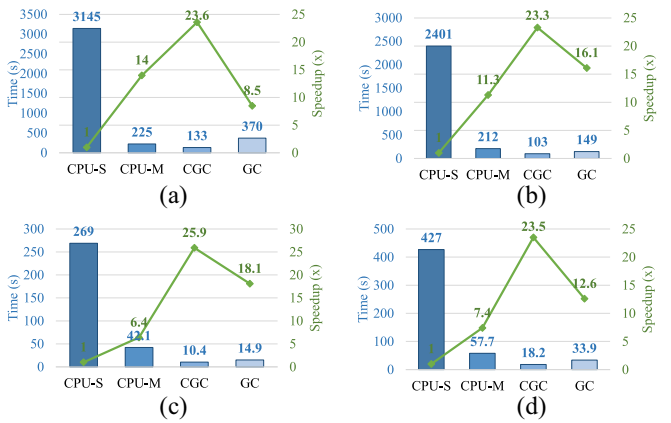

Fig. 16. Simulation time under different methods (single CPU and single GPU). CPU-S: CPU single-threaded simulation. CPU-M: CPU multithreaded simulation (32 threads, ratio = 1). CGC: two-stage fault simulation (32 CPU threads). For b18*, ratio = 0.5; for b19*, ratio = 0.4; for netcard*, ratio = 0.2; and for leon2*, ratio = 0.2. GC: two-stage fault simulation under ratio = 0 (32 CPU threads). (a) b18*. (b) b19*. (c) netcard*. (d) leon2*.

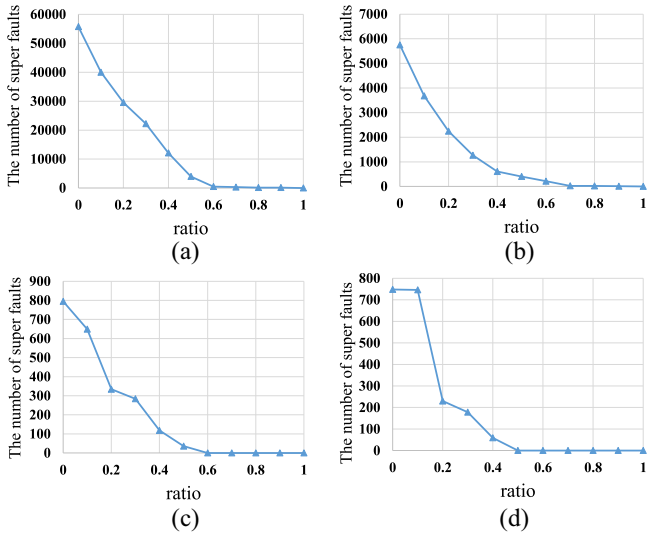

Fig. 17. Relationship between ratio and the number of super faults, where ratio represents the proportion of simulated faults on the CPU. (a) b18*. (b) b19*. (c) netcard*. (d) leon2*.

allocation ratio. GC is the method when the ratio is 0, that is, in the first stage, all faults are simulated on the GPU. We can observe that, compared with CPU-S, CGC can speed up 23.3×–25.9×. On different datasets, the acceleration of CGC is higher than that of CPU-M and GC.

2) Impact of Different Fault Allocation Ratios on the Results: Fig. 17 shows the effect of different ratios on the number of super faults. Super faults represent faults that cannot be detected by the GPU in the first stage of CGC. In other words, the number of active gates of super faults during simulation exceeds the maximum length of the simulation priority queue. We can see that as the ratio increases, that is, the proportion of simulated faults on the GPU decreases, the number of super faults also decreases. This can cause the CPU simulation time of the second stage to decrease. However, due to the different ratios of the CPU and GPU simulation faults in the first stage, the simulation time of the two can be inconsistent.

The influence of different ratios on the total time acceleration of CGC is shown in Fig. 18. We can observe that as

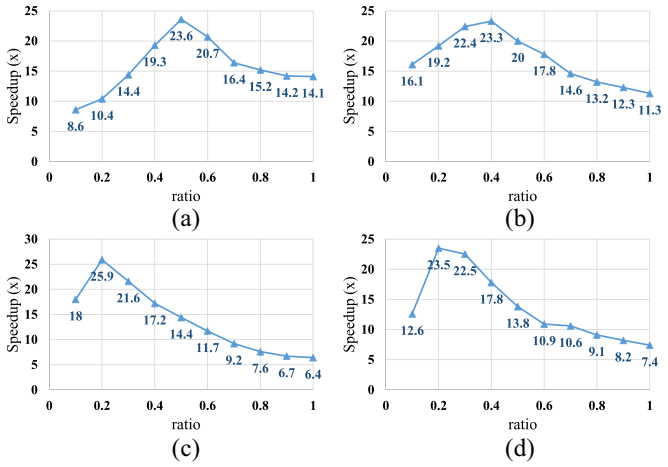

Fig. 18. Effect of ratio on acceleration factor (compared with CPU single-threaded simulation), where ratio represents the proportion of simulated faults on the CPU. (a) b18*. (b) b19*. (c) netcard*. (d) leon2*.

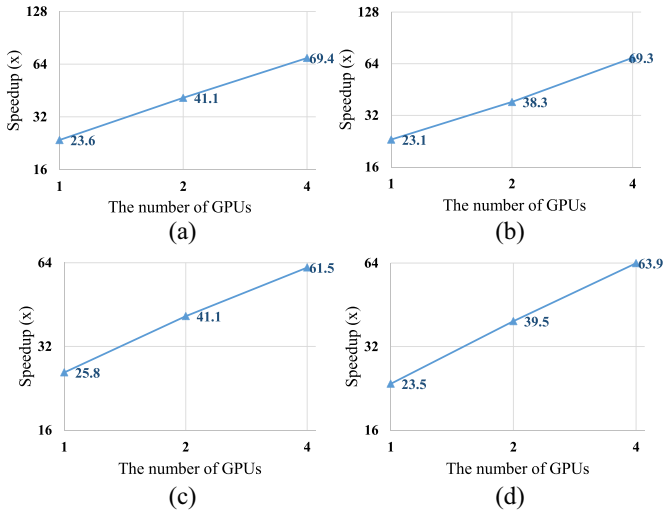

Fig. 19. GPU scalability. The acceleration factor changes with the number of GPU cards. (a) b18*. (b) b19*. (c) netcard*. (d) leon2*.

the ratio increases, the acceleration factor first increases and then decreases. When ratio = 0, in the first stage, all faults are simulated on the GPU. As the ratio increases, the simulation time of the CPU and GPU gradually reaches a balance, thereby accelerating the increase in multiples. However, with the further increase of the ratio, the number of faults in the CPU simulation is significantly greater than that of the GPU, thereby reducing the acceleration effect.

The scalability results on GPU and CPU are shown in Figs. 19 and 20, respectively. For GPU, the acceleration factor is almost linear with the number of GPU cards. In the case of using four GPU cards, the simulation time of CGC on each dataset can be accelerated by more than 60× compared with the CPU single-threaded simulation. For the CPU, Fig. 20 shows the relationship between the acceleration factor at different ratios and the number of CPU threads. There are obvious differences in the scalability of CPUs on different datasets. In summary, under the same ratio, the accelerating trend is first fast and then slow. The reason is that the increase of CPU threads introduces additional overhead so that the acceleration factor cannot be increased linearly.

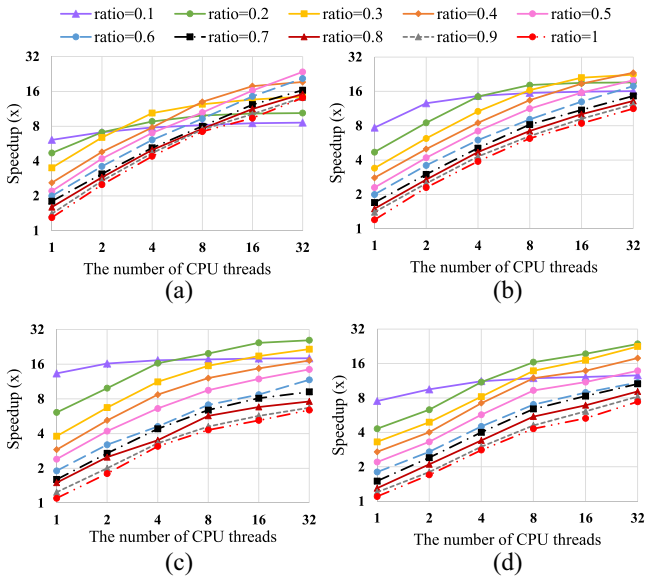

Fig. 20. CPU scalability. The acceleration factor changes with the number of CPU threads under different ratios. (a) b18*. (b) b19*. (c) netcard*. (d) leon2*.

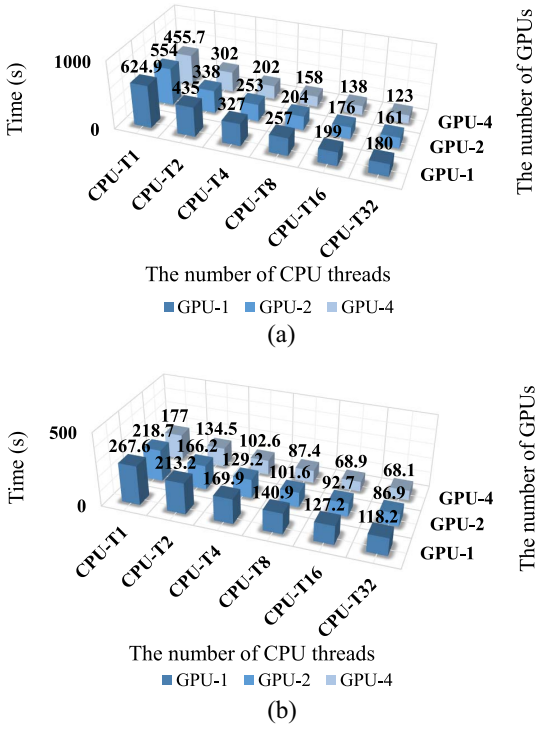

Fig. 21. 3-D figure of the simulation time of CGC with the number of CPU threads and the number of GPUs. Among them, CPU-Tx represents that the number of threads used in CPU simulation is x. GPU-x represents x cards used in GPU simulation. The above results are all simulation results under the optimal fault allocation ratio. (a) b18*. (b) b19*.

Fig. 21 shows the 3-D figure of the simulation time of CGC with the number of CPU threads and the number of GPU cards. We can see that on different data sets, the simulation time decreases as the number of CPU threads increases. The simulation time also decreases as the number of GPU cards increases. However, on different data sets, the sensitivity of the simulation time to the two is different, so the trend of change is also different.

TABLE V
OPTIMAL RATIO OF CGC WITH THE NUMBER OF CPU THREADS AND THE NUMBER OF GPUS UNDER B19* AND NETCARD*

	CPU Threads	T1	T2	T4	T8	T16	T32
b19*	GPU-1	0	0.1	0.1	0.2	0.3	0.3
	GPU-2	0	0	0.1	0.1	0.2	0.2
	GPU-4	0	0	0	0.1	0.1	0.1
netcard*	GPU-1	0.1	0.1	0.1	0.1	0.2	0.2
	GPU-2	0	0.1	0.1	0.1	0.1	0.2
	GPU-4	0	0.1	0.1	0.1	0.1	0.1

TABLE VI
RELATIONSHIP BETWEEN CIRCUIT DEPTHS AND OPTIMAL RATIOS

Dataset	**netcard***	**leon2***	**b19***	**b18***
Circuit Depth	72	100	460	673
Optimal Ratio	0.2	0.2	0.4	0.5

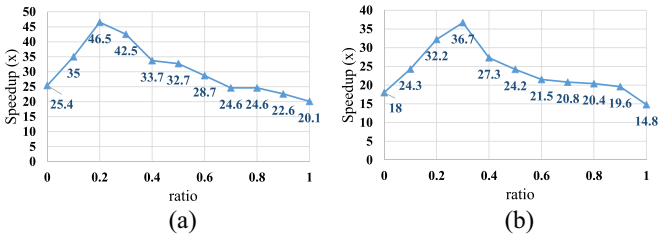

Fig. 22. Effect of ratio on acceleration factor (compared with CPU single-threaded simulation), where ratio represents the proportion of simulated faults on the CPU. (a) designA. (b) designB.

Table V shows the 3-D figure of the change of the optimal fault allocation ratio under different CPU threads and different GPU cards. Under the premise that the number of GPU cards remains unchanged, the ratio increases with the increase of the number of CPU threads. Under the premise that the number of CPU threads remains unchanged, the ratio decreases with the increase of the number of GPU cards. On the whole, the changing trends of the optimal ratios on different datasets are not the same. The main reason is that under different circuit structures, the ratio is different in sensitivity to the number of CPU threads and the number of GPU cards. In other words, under the same environment, there is a difference in the speed ratio of the GPU and the CPU simulating the same fault.

3) Influence of Circuit Depth on Optimal Ratio: Table VI lists the circuit depths and the corresponding optimal ratios for the four datasets in this article. It can be observed that as the depth of the circuit increases, the optimal ratio also increases gradually, that is, the number of faults allocated to the CPU increases. The reason is because when the length of the simulation queue is constant (set to 10 000 in our experiments), as the circuit depth increases, the length of the overall simulation path also increases. It causes the GPU to have more faults that cannot be simulated. Therefore, the allocation ratio on the GPU needs to be reduced.

G. Performance Evaluation for Industrial Circuits

We also test the performance of the CGC algorithm on actual large-scale industrial circuits. The hardware configuration in our experiments is Intel Xeon Gold 6140 2.30-GHz CPU and NVIDIA Tesla V100 GPU with 32-GB memory.

Under different ratios, the acceleration effect compared to the single-threaded CPU is shown in Fig. 22. For two industrial circuits, when using a GPU and 32 CPU threads, the CGC can reach 46.5× and 36.7× as high as possible. The

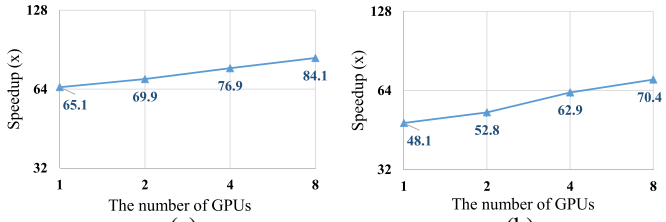

Fig. 23. GPU scalability. The acceleration factor changes with the number of GPU cards. (a) designA. (b) designB.

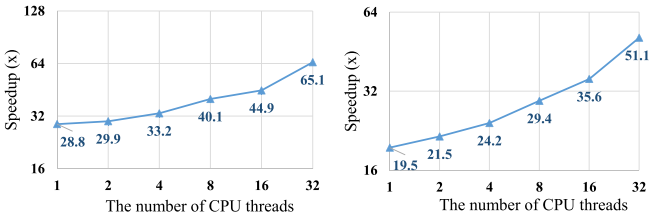

Fig. 24. CPU scalability. The acceleration factor changes with the number of CPU threads under different ratios. (a) designA. (b) designB.

scalability of the CPU and GPU is shown in Figs. 23 and 24, respectively. Among them, the multi-GPU has a sublinear situation, and the multi-CPU thread has a super-linear situation. The reason is that under different hardware resources, the fault distribution ratio is different.

VII. Conclusion

In this article, we improve the performance of fault simulation based on the CPU–GPU heterogeneous platform under different circuit scales. For small-scale circuits, we propose a novel fault grouping strategy based on the FFR. On this basis, we also use unified FR simulation to reduce code divergence and improve the simulation performance. For large-scale circuits, we propose different data structures for different hardware to reduce memory. Further, we propose a two-stage heterogeneous simulation algorithm CGC. For circuits of different scales, we propose two 4-D simulation architectures DFBP and DFBG to further improve their scalability and parallelism. The results show that our fault simulator for small-scale circuits based on a single GPU is 22× faster than the commercial tool on average, and based on 8 GPUs is 105.7× on average. Our fault simulator for large-scale circuits based on a GPU is up to 25.9× faster than the CPU single-threaded simulator.

References

[1] X. Cai and P. Wohl, "A distributed-multicore hybrid ATPG system," in *Proc. IEEE Int. Test Conf.*, 2013, pp. 1–7.

[2] M. Abramovici, M. A. Breuer, and A. D. Friedman, *Digital Systems Testing and Testable Design*. New York, NY, USA: IEEE Press, 1990.

[3] M. B. Amin and B. Vinnakota, "Workload distribution in fault simulation," *J. Electron. Test.*, vol. 10, no. 3, pp. 277–282, 1997.

[4] A. Abramovici, Y. H. Levendel, and P. R. Menon, "A logic simulation machine," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 2, no. 2, pp. 82–94, Apr. 1983.

[5] P. Agrawal, W. J. Dally, W. C. Fischer, H. V. Jagadish, A. S. Krishnakumar, and R. Tutundjian, "Mars: A multiprocessor-based programmable accelerator," *IEEE Design Test*, vol. 4, no. 5, pp. 28–36, Oct. 1987.

[6] R. Mueller-Thuns, D. G. Saab, R. F. Damiano, and J. A. Abraham, "VLSI logic and fault simulation on general-purpose parallel computers," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 12, no. 3, pp. 446–460, Mar. 1993.

[7] V. Narayanan and V. Pitchumani, "Fault simulation on massively parallel SIMD machines algorithms, implementations and results," *J. Electron. Test. Theory Appl.*, vol. 3, no. 1, pp. 79–92, 1992.

[8] S. Parkes, P. Banerjee, and J. Patel, "A parallel algorithm for fault simulation based on PROOFS," in *Proc. Int. Conf. Comput. Des.*, 1995, pp. 616–621.

[9] S. Patil and P. Banerjee, "Performance trade-offs in a parallel test generation/fault simulation environment," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 10, no. 12, pp. 1542–1558, Dec. 1991.

[10] S. Tai and D. Bhattacharya, "Pipelined fault simulation on parallel machines using the circuit flow graph," in *Proc. Int. Conf. Comput. Des.*, 1993, pp. 564–567.

[11] M. B. Amin and B. Vinnakota, "Data parallel fault simulation," in *Proc. Int. Conf. Comput. Des.*, 1995, pp. 610–615.

[12] N. Ishiura, M. Ito, and S. Yajima, "Dynamic two-dimensional parallel simulation technique for high-speed fault simulation on a vector processor," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 9, no. 8, pp. 868–875, Aug. 1990.

[13] F. Ozguner, C. Aykanat, and O. Khalid, "Logic fault simulation on a vector hypercube multiprocessor," in *Proc. Conf. Hypercube Concurrent Comput. Appl.*, 1988, pp. 1108–1116.

[14] F. Ozguner and R. Daoud, "Vectorized fault simulation on the Cray XMP supercomputer," in *Proc. Int. Conf. Comput.-Aided Des.*, 1988, pp. 198–201.

[15] N. Mittal and S. Kumar, "Machine Learning computation on multiple GPU's using CUDA and message passing interface," in *Proc. IEEE 2nd Int. Conf. Power Energy Environ. Intell. Control (PEEIC)*, 2019, pp. 18–22.

[16] P. Li, Y. Luo, N. Zhang, and Y. Cao, "HeteroSpark: A heterogeneous CPU/GPU Spark platform for machine learning algorithms," in *Proc. IEEE Int. Conf. Netw. Archit. Storage (NAS)*, Boston, MA, USA, 2015, pp. 347–348.

[17] S. W. Seo, J. Kyong, and E.-J. Im, "Social network analysis algorithm on a many-core GPU," in *Proc. 4th Int. Conf. Ubiquitous Future Netw. (ICUFN)*, Phuket, Thailand, 2012, pp. 217–218.

[18] X. Liu, M. Li, S. Li, S. Peng, X. Liao, and X. Lu, "IMGPU: GPU-accelerated influence maximization in large-scale social networks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 1, pp. 136–145, Jan. 2014.

[19] K. Gulati and S. P. Khatri, "Towards acceleration of fault simulation using graphics processing units," in *Proc. 45th ACM/IEEE Des. Autom. Conf.*, 2008, pp. 822–827.

[20] M. A. Kochte, M. Schaal, H.-J. Wunderlich, and C. G. Zoellin, "Efficient fault simulation on many-core processors," in *Proc. Des. Autom. Conf.*, 2010, pp. 380–385.

[21] H. K. Lee and D. S. Ha, "An efficient, forward fault simulation algorithm based on the parallel pattern single fault propagat," in *Proc. IEEE Int. Test Conf.*, 1991, pp. 946–955.

[22] D. Chatterjee, A. DeOrio, and V. Bertacco, "GCS: High-performance gate-level simulation with GPGPUS," in *Proc. Des. Autom. Test Eur. Conf.*, 2009, pp. 1332–1337.

[23] D. Chatterjee, A. DeOrio, and V. Bertacco, "Event-driven gate-level simulation with GP-GPUS," in *Proc. Des. Autom. Conf.*, 2009, pp. 557–562.

[24] M. Li and M. S. Hsiao, "3-D parallel fault simulation with GPGPU," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 30, no. 10, pp. 1545–1555, Oct. 2011.

[25] L. Lai, K.-H. Tsai, and H. Li, "GPGPU-based ATPG system: Myth or reality?" *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 1, pp. 239–247, Jan. 2020.

[26] "NVIDIA CUDA homepage." Accessed: Sep. 2021. [Online]. Available: http://www.nvidia.com/object/cudahome.html

[27] "ITC'99 benchmarks." Accessed: Sep. 2021. [Online]. Available: http://www.cad.polito.it/downloads/tools/itc99.html

[28] "Virginia tech VLSI for telecommunications." Accessed: Sep. 2021. [Online]. Available: http://www.vtvt.ece.vt.edu/vlsidesign/cadtools.php

[29] H. S. Lee, "IC design challenges and opportunities for advanced process technology," in *Proc. VLSI Des. Autom. Test(VLSI-DAT)*, Hsinchu, Taiwan, 2015, pp. 1–2.

[30] B. Mitra, "Consumer digitization: Accelerating DSP applications, growing VLSI design challenges," in *Proc. ASP-DAC/VLSI Des. 7th Asia South Pac. Des. Autom. Conf. 15th Int. Conf. VLSI Des.*, Bangalore, India, 2002, pp. 3–4.

[31] B. J. Lavanyashree and S. Jamuna, "Design of fault injection technique for VLSI digital circuits," in *Proc. 2nd IEEE Int. Conf. Recent Trends Electron. Inf. Commun. Technol. (RTEICT)*, Bangalore, India, 2017, pp. 1601–1605.
[32] "IWLS 2005 benchmarks." 2005. [Online]. Available: http://iwls.org/iwls2005/benchmarks.html
[33] J. G. Tong, M. Boulé, and Z. Zilic, "Efficient data encoding for improving fault simulation performance on GPUs," in *Proc. Int. Symp. Electron. Syst. Des.*, 2013, pp. 138–142.
[34] H. Li, D. Xu, Y. Han, K.-T. Cheng, and X. Li, "nGFSIM: A GPU-based fault simulator for 1-to-*n* detection and its applications," in *Proc. IEEE Int. Test Conf.*, 2010, pp. 1–10.

Jingbo Hu (Student Member, IEEE) received the B.S. degree from the Department of Communication Engineering, Xidian University, Xi'an, China, in 2015 and 2019, respectively. She is currently pursuing the master's degree with the Department of Electronic Engineering, Tsinghua University, Beijing, China.

Her current research interests include graph computing, VLSI SoC testing, ATPG, vector retrieval, and heterogeneous computing, especially the efficient processing of very large-scale data in heterogeneous systems.

Guohao Dai (Member, IEEE) received the B.S. and Ph.D. (with Hons.) degrees from Tsinghua University, Beijing, China, in 2014 and 2019, respectively.

He is an Associate Professor with Shanghai Jiao Tong University, Shanghai, China. His research mainly focuses on large-scale sparse graph computing, heterogeneous hardware computing, and emerging hardware architecture.

Dr. Dai has received the Best Paper Award in ASP-DAC 2019 and the Best Paper Nomination in DAC 2022 and DATE 2018. He is the winner of the NeurIPS Billion-Scale Approximate Nearest Neighbor Search Challenge in 2021 and the recipient of the Outstanding Ph.D. Dissertation Award of Tsinghua University in 2019. He currently serves as the PI/Co-PI for several projects with a personal share of over RMB 6 million.

Liuzheng Wang received the B.S. degree and the M.S. degree in computer science from the Huazhong University of Science and Technology, Wuhan, China, in 2008 and 2011, respectively.

From 2011 to 2020, he was with the Design-for-Test Division, HiSilicon, Shenzhen, China. Since 2021, he has been an Architect of EDA software. His current research interests include graph computing and heterogeneous computing.

Liyang Lai (Member, IEEE) received the B.S. degree in microelectronics from Peking University, Beijing, China, in 1997, the M.S. degree in electrical engineering from the Institute of Microelectronics, Chinese Academy of Sciences, Beijing, in 2000, and the Ph.D. degree in electrical engineering from the University of Illinois at Urbana–Champaign, Champaign, IL, USA, in 2005.

He is an Associate Professor with the Department of Electrical Engineering, Shantou University, Shantou, Guangdong, China. He worked with the Silicon Test Division, Mentor Graphics, Wilsonville, OR, USA, from 2005 to 2013 and was a Consulting Staff with Calypto Design Systems, Wilsonville, from 2013 to 2015. His research interests include VLSI design and test, fault-tolerant computing, computer architecture, and high-level synthesis.

Dr. Lai is a coauthor of the 2006 ITC Best Paper.

Yu Huang (Senior Member, IEEE) received the Ph.D. degree in electrical and computer engineering from the University of Iowa, Iowa City, IA, USA, in 2002.

He is the EDA Chief Architect of HiSilicon, Shenzhen, China. Before Joining HiSilicon, he was a Senior Key Expert of Mentor Graphics, Wilsonville, OR, USA. He has more than 70 patents and published more than 130 papers on leading IEEE journals, conferences, and workshops. His research interests include VLSI SoC testing, ATPG, compression, diagnosis, yield analysis, machine learning, and AI chips.

Dr. Huang has served as a Technical Program Committee Member for DAC, ITC, VTS, ATS, ETS, ASPDAC, NATW, and many other conferences and workshops in the testing area. He is an Invited Reviewer of many leading IEEE journals, such as IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS and IEEE TRANSACTIONS ON COMPUTERS.

Huazhong Yang (Fellow, IEEE) received the B.S. degree in microelectronics and the M.S. and Ph.D. degrees in electronic engineering from Tsinghua University, Beijing, China, in 1989, 1993, and 1998, respectively.

In 1993, he joined the Department of Electronic Engineering, Tsinghua University, where he has been a Professor since 1998. He has been in charge of several projects, including projects sponsored by the National Science And Technology Major Project, 863 Program, NSFC, and several international research projects. He has authored and coauthored over 500 technical papers, seven books, and over 180 granted Chinese patents. His current research interests include wireless sensor networks, data converters, energy-harvesting circuits, nonvolatile processors, and brain-inspired computing.

Prof. Yang was awarded the Distinguished Young Researcher by NSFC in 2000, the Cheung Kong Scholar by the Chinese Ministry of Education (CME) in 2012, the Science and Technology Award First Prize by China Highway and Transportation Society in 2016, and the Technological Invention Award First Prize by CME in 2019. He has also served as the Chair of Northern China ACM SIGDA Chapter Science 2014, the General Co-Chair of ASPDAC20, a Navigating Committee Member of AsianHOST18, and a TPC Member for ASPDAC05, APCCAS06, ICCCAS07, ASQED09, and ICGCS10.

Yu Wang (Fellow, IEEE) received the B.S. and Ph.D. (with Hons.) degrees from Tsinghua University, Beijing, China, in 2002 and 2007, respectively.

He is currently a Tenured Professor with the Department of Electronic Engineering, Tsinghua University. He has authored and coauthored more than 200 papers in refereed journals and conferences. His research interests include brain-inspired computing, application-specific hardware computing, parallel circuit analysis, and power/reliability-aware system design methodology.

Dr. Wang has received the Best Paper Award in ASPDAC 2019, FPGA 2017, NVMSA 2017, and ISVLSI 2012, and the Best Poster Award in HEART 2012 with nine Best Paper Nominations (DATE18, DAC17, ASPDAC16, ASPDAC14, ASPDAC12, 2 in ASPDAC10, ISLPED09, and CODES09). He is a recipient of the DAC Under 40 Innovator Award in 2018 and the IBM X10 Faculty Award in 2010. He served as the TPC Chair for ICFPT 2019 and 2011, and ISVLSI2018, the Finance Chair of ISLPED 2012–2016, and the Track Chair for DATE 2017–2019 and GLSVLSI 2018, and served as a Program Committee Member for leading conferences in these areas, including top EDA conferences, such as DAC, DATE, ICCAD, and ASP-DAC, and top FPGA conferences, such as FPGA and FPT. He currently serves as the Co-Editor-in-Chief for the ACM SIGDA E-Newsletter, an Associate Editor for the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS, the IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS FOR VIDEO TECHNOLOGY, and the *Journal of Circuits, Systems, and Computers*, and the Special Issue Editor for the *Microelectronics Journal*. He is currently with ACM Distinguished Speaker Program.