Simulation-based Parallel Sweeping: A New Perspective on Combinational Equivalence Checking

Tianji Liu
CSE Department, CUHK
tjliu@cse.cuhk.edu.hk

Evangeline F.Y. Young
CSE Department, CUHK
fyyoung@cse.cuhk.edu.hk

***Abstract*—Combinational equivalence checking (CEC) is a fundamental task in the realization of digital designs which is unlikely to have universally efficient algorithms due to its co-NP-completeness. Recent researches of CEC have been focusing on SAT sweeping. This paper provides a new perspective other than SAT for tackling CEC, namely exhaustive simulation, and presents a simulation-based CEC engine constructed with fast GPU-parallel algorithms. The proposed engine can solve 4 out of the 9 large cases in the experiments on its own, with up to 88.11× speed-up compared with the checker in ABC. Moreover, a combination of the proposed engine with the ABC checker achieves averaged accelerations of 4.89× and 4.88× over the standalone ABC checker and a commercial checker, respectively.**

I. Introduction

Combinational equivalence checking (CEC) is one of the most important and fundamental problems in the realization flow of modern digital designs, in which the objective is to prove the functional equivalence of two netlists that implement the same combinational circuit differently. Besides logic verification [1], CEC approaches also have a wide range of applications in the areas including logic synthesis [2], [3] and functional ECO [4].

Many tasks in EDA can be formulated as optimization problems that aim to reduce certain costs, e.g., area and delay reduction in logic optimization and technology mapping, and wirelength reduction in placement and routing, which enables the use of approximate and heuristic approaches that run in polynomial time, making them scalable to large designs. In contrast, CEC requires an exact answer of whether two circuits are equivalent, i.e., the problem needs to be tackled directly without any approximation, and it is thus much more challenging than the aforementioned tasks considering the complexity of CEC as well as the increasing scale of modern designs.

Since CEC is co-NP-complete, efficient verification of all kinds of designs using currently developed approaches is essentially infeasible. Binary decision diagram (BDD) [5], [6] had been a common method for CEC in the early years, but was gradually replaced by Boolean satisfiability (SAT) solving [7], [8] due to the excessive memory consumption of BDD which impedes its application to large designs. Today, SAT has become the de facto core technique in CEC because of the noticeable performance improvement in SAT solving over the past two decades [9]–[11]. Nevertheless, it has been argued that SAT-based methods are not as performant as other approaches for certain types of designs, e.g., computer algebra methods for arithmetic circuits [12]. This implies that the demand for novel and fast CEC approaches will never diminish, as long as new and larger sized designs continue to emerge.

In this paper, we present a new perspective on tackling CEC, namely utilizing *exhaustive simulation* for proving functional equivalences. A key advantage of exhaustive simulation over SAT is that it is friendly to parallelization, and we will demonstrate in this paper that exhaustive simulation can be performed efficiently on GPUs, which are powerful massively parallel processors that have shown promising results in logic synthesis applications [13], [14]. When proving difficult cases, however, parallel exhaustive simulation may still be intractable due to its intrinsic exponential complexity. To address this issue, we introduce a local function checking scheme for restricting the computational effort of checking difficult cases, and the effectiveness of checking can still be ensured by delicately designed heuristics. We propose and implement fast GPU-parallel algorithms for exhaustive simulation, local function checking and other auxiliary procedures for CEC, which are combined together as a *simulation-based CEC engine* with high overall performance.

The simulation-based CEC engine is capable of independently proving 4 out of the 9 large circuits in the experiments, with up to 88.11× acceleration compared to the SAT-based equivalence checker in ABC [15]. When combined with the ABC checker for handling the undecided cases, the integrated approach obtains 4.89× and 4.88× speed-up on average compared to the standalone ABC checker and a commercial verification tool, respectively.

This research was partially supported by a grant from the Research Grants Council of the Hong Kong Special Administrative Region, China (Project No. CUHK14210923).

II. Preliminaries

A. Background

A *Boolean network* is a directed acyclic graph (DAG) that models a combinational circuit where nodes stand for logic operations and edges represent wires between the nodes. The *primary inputs* (PIs) and *primary outputs* (POs) of a Boolean network are the sources and the sinks of the DAG. The direct predecessors (resp. direct successors) of a node are called the *fanins* (resp. *fanouts*) of the node. The predecessors (resp. successors) of a node are called the *transitive fanins* (TFIs) (resp. *transitive fanouts*, TFOs) of the node. The *(structural) support* of a node denotes the set of PIs that are TFIs of the node. The function of a node expressed in terms of its support is the *global function* of the node. The *level of a node* is the length of a longest path from any PI to the node, which can be computed recursively as the maximum level of the node's fanins plus one. The *level of the network* is the largest level of its POs.

A *cut* c_n of a node n is a set of nodes such that any path from a PI to n passes at least one node in the set, and n is the *root* of c_n. The *trivial cut* of n is defined as $\{n\}$. The function of a node in terms of its cut nodes is the *local function* of the node. The *logic cone* of a node n (associated with a cut c_n) contains the intersection of the TFIs of n with the TFOs of c_n, and n itself.

An *And-Inverter Graph* (AIG) [16] is a Boolean network in which nodes are two-input AND gates and signals (edges) can be optionally inverted. Because of its compactness and flexibility, AIG has become one of the most common circuit representations in modern logic synthesis and verification tools such as ABC [15]. In this work, we

also focus on the verification of AIGs but the proposed methods can be applied to any kind of circuit representation.

The *truth table* of a k-input Boolean function is a bit string $b_{l-1} \dots b_0$ of length $l = 2^k$, in which a bit $b_i \in \{0, 1\}$ is the function value under the assignment of the k inputs $(a_0, \dots, a_{k-1})$ that satisfies $2^{k-1} \cdot a_{k-1} + \dots + 2^0 \cdot a_0 = i$. Essentially, a truth table consists of the function values under all possible input assignments[1]. We denote the truth tables of projection functions $f_i(x_0, \dots, x_{k-1}) = x_i$ as *projection truth tables*, which serve as the truth tables of inputs when computing the truth table of a node in a network. For instance, when $k = 3$, the projection truth tables of f_0, f_1, f_2 are 10101010, 11001100, 11110000.

The *(internal) satisfiability don't cares* (SDCs) of a node in a Boolean network are the patterns at a cut of the node that will never occur. For example, in the network with nodes $n_1 = x + y$, $n_2 = yz$ and $n_3 = n_1 n_2$, n_3 has SDCs of $\{(n_1 = 0, n_2 = 1)\}$. If a node n has SDCs, the local function values of n under the SDC assignments can be changed without affecting the global function of n. It is believed that SDCs are mainly due to reconvergent paths (different paths with the same start and end) in the TFI structure of the cut [17], [18].

B. Sweeping

In CEC, a *miter* [19] circuit is constructed by sharing the corresponding PI pairs of the two circuits being compared, and combining the corresponding PO pairs using XOR gates which are treated as the POs of the miter. Then, the problem of checking whether the two circuits are equivalent becomes checking whether all the POs of the miter are constant zeros.

A common and effective framework for CEC is *sweeping* [6], [16]. The basic idea of sweeping is to identify equivalent pairs of internal nodes in the miter and merge their logic, so the miter can be gradually reduced, making it easier to prove the POs of the miter. A fast *(partial) simulation* procedure [16] is performed in the beginning of sweeping, which computes the partial truth tables of all the nodes using simulation patterns assigned to the PIs (usually randomly generated), and clusters the nodes with the same partial truth table into an *equivalence class* (EC). This reduces the number of candidate node pairs that need formal checking, since any pair of equivalent nodes must reside in the same class. A usual way to generate candidate pairs is to take the *representative* of a class (defined as the node with the minimum id in the class) and match it with the non-representatives in the class, so a class of N nodes produces $N - 1$ pairs. The equivalence checking of candidate pairs is performed using a formal method, e.g., BDD [6] or SAT [8].

Recent researches mainly focus on improving the SAT sweeping framework. In [3], [20], [21], different methods are proposed for generating high-quality simulation patterns to reduce SAT calls. [21], [22] introduce a customized SAT solver with engineering efforts tailored for SAT sweeping. There have also been works on accelerating SAT sweeping via parallelization. [23] describes a method for concurrently executing SAT checks for multiple candidate node pairs. [24] develops a hybrid CPU-GPU framework in which GPU performs simulation and local matching of candidate pairs, and the pairs undetermined in local matching are further checked by SAT on CPU. The proposed CEC engine drastically differs from these methods in that it relies on exhaustive simulation rather than SAT for proving equivalent nodes.

[1]We also use the term *patterns* for input assignments in this paper.

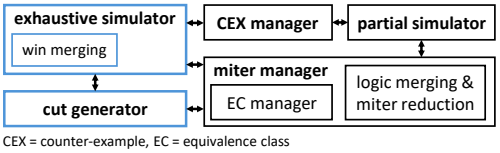

Fig. 1. The structure of the simulation-based CEC engine. The core modules are highlighted in blue. Arrows indicate interactions between modules.

III. Simulation-based CEC Engine

This section introduces the simulation-based CEC engine. An overview of the CEC engine is first provided in Section III-A, followed by two subsections elaborating on the exhaustive simulator and local function checking, which are core components of the engine. Finally, the overall flow of the engine is presented in Section III-D.

A. Overview of Simulation-based CEC Engine

The simulation-based CEC engine consists of five modules, as depicted in Figure 1, in which the exhaustive simulator and cut generator are the core of the engine. The exhaustive simulator serves as the prover in the engine that checks whether a candidate pair of nodes is equivalent by comparing all the possible values in their entire truth tables. An input pattern that disproves the candidate pair, i.e., yielding different function values at the two nodes, will be collected as a *counter-example* (CEX). The exhaustive simulator is designed to be efficient: it is capable of checking a batch of candidate pairs using a highly parallel algorithm, with a window merging procedure for reducing the total simulation effort.

The cut generator plays a crucial role in checking candidate pairs of nodes with large support sizes. Due to exponential complexity, it is infeasible to exhaustively simulate the global functions of such nodes. Instead, the cut generator produces multiple common cuts for each pair of nodes, and the exhaustive simulator evaluates the corresponding local functions of the candidate pairs. By limiting the sizes of these cuts, the simulation effort can be successfully controlled. A detailed introduction to cut generation will be given in Section III-C1.

The remaining modules have their counterparts in the traditional SAT sweeping framework which are conceptually similar to them. The miter manager maintains the miter in AIG data structures and can reduce the miter by merging proved pairs of equivalent nodes. It is also equipped with an EC manager which is responsible for maintaining the equivalence class information and generating candidate pairs to be proved. The partial simulator performs simulation of random patterns for initializing the ECs, as well as CEX patterns for splitting the class of a disproved pair (and potentially other classes) into smaller ones.

All the modules and subroutines in the simulation-based CEC engine are GPU-based except window merging which is partially implemented on CPU, making the engine highly efficient.

B. Exhaustive Simulator

1) Equivalence Checking via Exhaustive Simulation: We start by introducing how to check the equivalence of a pair of nodes via exhaustive simulation. The basic idea, as mentioned in Section III-A, is to compare the entire truth tables of the two nodes. However, the comparison is only meaningful if the input variables (including their ordering) of the two truth tables are exactly the same. Consider the function $xy' + xy'z$ as an example. Its truth tables are 00100010 and 01000100 for the input orders (x, y, z) and (y, x, z) respectively.

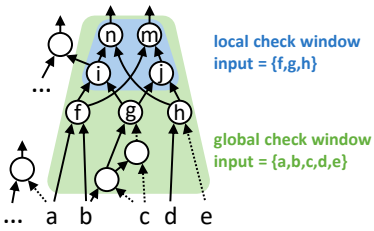

Fig. 2. Illustration of windows in global and local function checking of two nodes n and m. Note, their local functions in terms of $\{f, g, h\}$ are equivalent.

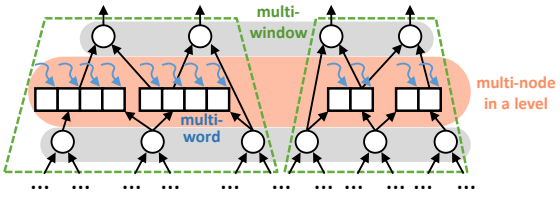

Fig. 3. Illustration of the three dimensions of parallelism.

Also, note that $xy' + xy'z = xy'$, and the truth table of xy' with input order (x, y) is 0010, indicating that the truth table of equivalent functions may not be the same under different variables and orderings.

The proposed exhaustive simulator handles this issue as follows. In the case of simulating the global functions of the two nodes being checked, if their support sets are not identical, the union of their supports are taken as the input nodes of the truth table with the order of increasing node ids. Since it is possible that a node does not functionally depends on one or more of its support nodes (e.g., $xy' + xy'z$ does not functionally depend on z), two nodes with nonidentical supports can be equivalent and still need to be checked. In the case of checking the local functions of a node pair, the inputs are simply the nodes in the corresponding common cut of the pair provided by the cut generator, as introduced in Section III-A.

The truth tables of two nodes n, m being checked are computed by propagating truth tables along the intermediate nodes that drive n and m, starting from the input nodes. We denote the collection of such intermediate nodes as the *simulation window* (or simply *window*) of n and m, and the two nodes are called the *roots* of the window. Figure 2 provides an illustration of windows in the scenario of simulating the global and local functions of two nodes respectively. Formally, a window contains the intersection of the TFIs of the roots with the TFOs of the inputs, as well as the roots.

The computation procedure begins with assigning projection truth tables to the input nodes. Then, the truth table of a node in the window can be derived by processing the truth tables of its fanins according to the node's operation. For instance, in the AIG shown in Figure 2, the truth table of the node $h = de'$ can be computed as $TT(h) = TT(d)$ & $!TT(e)$ where &, ! stand for bitwise AND, NOT respectively. The computations for all the nodes in the window should be scheduled in a topological order to ensure correctness. In our exhaustive simulator, this is performed in a level-wise parallel way, which is detailed in the next subsection.

2) Parallel Simulation: The exhaustive simulator is equipped with three dimensions of parallelism, as illustrated in Figure 3. The most fine-grained parallelism happens in the truth table computation of a single node, in which each word (a 32-bit or 64-bit element) in the bit string of the truth table is computed by a separate thread. Such parallel processing is particularly efficient on GPUs because of coalesced memory accesses [25] which minimizes bandwidth usage.

Algorithm 1 Parallel Exhaustive Simulation

Input: Miter, set of pairs p and windows w, available memory in words M

1: $N \leftarrow \sum_{w_i \in w}(|w_i| + |inputs(w_i)|)$ ▷ number of nodes and inputs
2: $E \leftarrow \max_{e \in \mathbb{N}} 2^e$ s.t. $2^e \cdot N \leq M$ ▷ entry size of simulation table
3: $simt \leftarrow$ simulation table containing $E \cdot N$ words
4: $r_m \leftarrow (\max_{w_i \in w} tt\_len(w_i))/E$ ▷ number of rounds
5: **for each** $r = 0, \ldots, r_m - 1$ **do** ▷ simulate range $[rE, (r+1)E)$
6: $w' \leftarrow \{w_i \in w : tt\_len(w_i) \geq (r+1)E\}$
7: ▷ w' is the set of windows that need simulation in round r
8: $l_m \leftarrow \max_{w_i \in w'} \max_{n \in w_i} level(n)$ ▷ max level in round r
9: **parallel** write proj. TT segments into $simt$ for inputs to all $w_i \in w'$
10: **for each** $l = 1, \ldots, l_m$ **do**
11: **parallel** simulate all nodes in w' with level l, and update $simt$
12: **parallel for each** $p_i = (n_i, m_i) \in p$ **do**
13: **if** the TT segments of n_i and m_i in $simt$ mismatch **then**
14: Mark p_i as nonequivalent, and optionally collect CEX

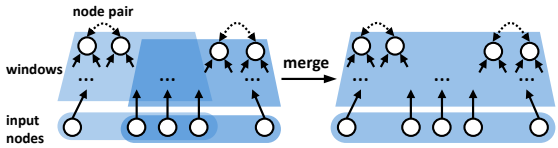

Fig. 4. Illustration of window merging.

The second dimension of parallelism resides in the truth table computation of the nodes in a window, following the level-wise parallel [14] fashion. Specifically, nodes with the same topological level are grouped into a batch and concurrently processed since there is no intra-batch dependency between nodes, and different batches are processed sequentially in the order of increasing topological level. The topological level of a node is similar to the node's level defined in Section II-A, and the difference is that the input nodes to the window are assigned with zero topological levels.

The third dimension of parallelism is to simulate multiple windows concurrently. If the number of inputs to a window is small, the truth tables to be computed are short in length, and the window is likely to be small sized. In this case, the degree of parallelism provided by the other two dimensions is limited, so it is essential to include this dimension to fully exploit the computational power of GPUs.

Algorithm 1 shows the high-level pseudocode of parallel exhaustive simulation. Given a batch of node pairs and their windows, a simulation table is allocated for storing the simulation results of all the nodes in the windows and the patterns at the window inputs (lines 1-3). Since the truth table of a pair can be long and there can be many pairs in the batch, it is impracticable for the simulation table to record the entire truth tables. Instead, each node or input is assigned with an entry of $E = 2^e$ words, where E is decided on-the-fly to be the maximum feasible value such that the simulation table fits in a provided size of memory (line 2). To ensure exhaustive simulation, the main procedure is formulated to be multi-rounded: in round r, the truth table words with indices in the range of $[rE, (r+1)E)$ are simulated with the three dimensions of parallelism (lines 6-11). The simulation results of all the node pairs are compared at the end of each round. If a mismatch is detected at a pair, this pair is marked as nonequivalent and the corresponding input pattern can be collected as a CEX (lines 12-14).

3) Reducing Simulation Effort via Window Merging: In exhaustive simulation, windows may overlap with each other and a node appearing in multiple windows must be simulated separately in each window due to nonidentical input nodes, which potentially affects the efficiency. A possible method for addressing this issue is to merge

multiple windows with high overlaps into one, and simulate all the corresponding pairs using the single merged window, as depicted in Figure 4. In this way, the total number of nodes that need simulation as well as the number of windows can be reduced. On the other hand, the number of inputs may increase after merging a window which leads to longer truth tables, and it should be carefully controlled to ensure that the overall effect of window merging is beneficial.

Our algorithm of window merging is performed in the beginning of exhaustive simulation and it employs a fast heuristic that works well in practice. First, we sort the batch of windows in the lexicographical order of their input nodes. Since the input set to a window is ordered in ids (Section III-B1), the windows with similar input sets tend to be closer to each other after sorting. The resulting windows are then generated by maximally merging consecutive windows in the order while keeping the input sizes of the merged windows under a threshold k_s. As an example, for the five windows with inputs $\{a,b\},\{a,b\},\{a,b,c\},\{a,e\},\{a,f\}$ and $k_s=3$, the first three and the last two are merged together respectively. We remark that better merging results may be obtained using a more dedicated approach (e.g., clustering-based) but it may induce a large overhead. Window merging is only enabled for global function checking, because it is generally not beneficial for local function due to small window sizes.

C. Local Function Checking

1) Cut Enumeration and Selection Criteria for Effective Checking: Local function checking is a key component in the simulation-based CEC engine, which is indispensable for proving the equivalences of node pairs with large support sizes that cannot be exhaustively simulated in terms of global functions. If the local functions of two nodes in terms of a common cut are identical, the two nodes are proved equivalent. As an example, the equivalence of the node pair shown in Figure 2 can be proved with the local functions in terms of the cut $\{f,g,h\}$, which reduces the computation from simulating 2^5 patterns per node in global function checking to 2^3 patterns.

However, two equivalent nodes may also have nonidentical local functions in terms of a common cut, e.g., those in terms of the cut $\{f,h,i,j\}$ in Figure 2. This is due to SDCs at the cut, and the equivalence is not affected as long as all the patterns yielding different local function values are SDCs. In the context of local function checking, the equivalence of two nodes is inconclusive if nonidentical local functions are encountered. To enhance the chance of successful checking, two strategies can be applied: increasing the quantity, and improving the quality of the common cuts used in checking.

To increase the quantity of cuts, we design a cut generator for the CEC engine that is capable of producing multiple common cuts for each pair using a cut enumeration-based [26] framework. For each AIG node n in the miter, the cut generator enumerates a set of candidate cuts $E(n)$ with maximum cut size k_l by

$$E(n) = \{u \cup v : u \in P(n_0) \cup \{\{n_0\}\}, \\ v \in P(n_1) \cup \{\{n_1\}\}, |u \cup v| \leq k_l\}, \quad (1)$$

where n_0, n_1 are the two fanins of n, and $P(\cdot)$ denotes the set of *priority cuts* [27] of the node that contains its best C candidate cuts selected using certain criteria. k_l and C are parameters that control the computational effort of local function checking. The common cuts of a pair can then be computed by Equation (1) with n_0 and n_1 replaced by the pair of nodes and without including their trivial cuts. The main advantages of cut enumeration are that it can efficiently produce multiple cuts for all the nodes (and common cuts for all the pairs) in a topological order traversal, and that it can adopt customized criteria for selecting a set of priority cuts.

TABLE I
CUT SELECTION CRITERIA IN DIFFERENT PASSES

Pass	Main Metric	Tie-breaker Metric 1	Tie-breaker Metric 2
1	fanout	cut size	small level
2	small level	cut size	fanout
3	large level	cut size	fanout

To obtain high-quality cuts, the cut selection criteria is formulated delicately in which the following metrics are involved:

- average number of fanouts of cut nodes: large fanout is used as a cutpoint selection heuristic in [16], so a high value is preferred;
- cut size: a small value is preferred for preventing oversized cuts during cut enumeration, and encouraging the inclusion of more reconvergent structures in the logic cone so that SDCs can be reduced, which enables higher chances of proving equivalences;
- average level of cut nodes: a small value is preferred for including more logic in the cone and reducing SDCs, but a high level may also be beneficial since it leads to smaller cuts, and sometimes a high-level cut is sufficient to prove a pair if one node is generated by the local restructuring of the other (e.g., Figure 2).

In practice, it is difficult to obtain cuts that are good in terms of all the metrics. We tackle this issue by applying three passes of cut generation and checking. In different passes, the metrics are considered with different priorities to increase the diversity of the generated cuts, as specified in Table I. For instance, in the first pass, cuts with larger fanouts are selected; if a tie happens, cuts of smaller size are preferred; if there is still a tie, cuts with smaller levels are used.

A problem of cut enumeration is that it lacks the view of common cuts when generating cuts for single nodes, which subsequently causes many oversized ($> k_l$) common cuts that cannot be used in local function checking. This problem is resolved by generating similar cuts for a pair of nodes in cut enumeration. Specifically, we define a similarity metric between a cut c and a set of priority cuts P as $s(c,P)=\sum_{c'\in P}|c\cap c'|/|c\cup c'|$. When enumerating the cuts of a non-representative node n, the similarity between a cut of n and the priority cut set of the corresponding representative node in the class is computed (since n and the representative form a pair), and cuts with larger similarities are preferred. In case of a tie, the criteria in Table I are used. On the other hand, the representative nodes adopt Table I for cut selection as introduced in the previous paragraph. In this way, the priority cuts of a pair of nodes tend to have high overlaps to each other, thus increasing the number of usable common cuts in checking.

2) Flow of Cut Generation and Checking Pass: Cut generation and checking are performed in level-wise parallel, where the level is specially formulated to consider the dependencies between representatives and non-representatives, since the priority cuts of a representative node should be computed before any other non-representatives in its class. We denote such levels as *enumeration levels*, defined by

$$el(n) = \begin{cases} 0, & n \text{ is PI}, \\ 1+\max\{el(n_0), el(n_1)\}, & n \text{ is not PI and } n \text{ is repr.}, \\ 1+\max\{el(n_0), el(n_1), el(repr(n))\}, & \text{otherwise}, \end{cases} \quad (2)$$

where $repr(n)$ signifies the corresponding representative of a non-representative node n, and n_0, n_1 are the two fanins of n.

Algorithm 2 describes the flow of a cut generation and checking pass. It starts with computing the enumeration levels of all the nodes (line 2) and assigning trivial cuts to the PIs as their priority cuts (lines 4-5). The priority cuts of internal nodes are then computed level by

Algorithm 2 Cut Generation and Checking Pass

Input: Miter $miter$

1: $buf \leftarrow \emptyset$ ▷ allocate common cut buffer
2: Compute enumeration levels $el(\cdot)$ of all nodes by Equation (2)
3: $l_m \leftarrow \max_{n \in miter} el(n)$
4: **parallel for each** PI n of $miter$ **do**
5: $P(n) \leftarrow \{\{n\}\}$
6: **for each** $l = 1, \ldots, l_m$ **do**
7: $enum \leftarrow \{n \in miter : el(n) = l\}$
8: ▷ $enum$ is the set of nodes whose cuts are to be computed at level l
9: **parallel for each** $n \in enum$ **do**
10: **parallel** compute $P(n)$ by cut enum. & selection (Section III-C1)
11: $p \leftarrow \{(repr(n), n) : n \in enum, n$ is a non-representative$\}$
12: **parallel** generate valid common cuts of pairs in p as cmn
13: **if** $|cmn| > capacity(buf) - |buf|$ **then** ▷ insufficient space
14: Local function checking using cuts in buf by Algorithm 1
15: $buf \leftarrow \emptyset$ ▷ clear the buffer
16: $buf \leftarrow buf \cup cmn$ ▷ insert in **parallel**
17: **if** $|buf| > 0$ **then** ▷ check the final batch
18: Local function checking using cuts in buf by Algorithm 1

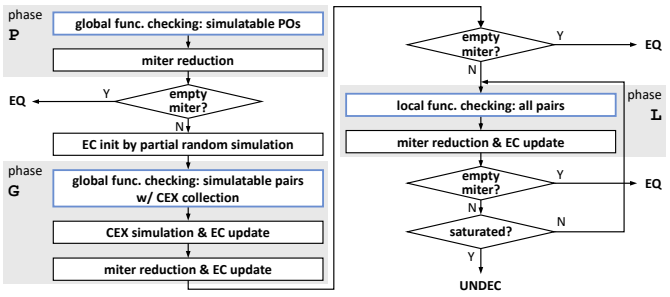

Fig. 5. Flow of simulation-based CEC engine.

level, following the criteria introduced in Section III-C1 (lines 6-10). In particular, cut enumeration and priority cut selection for a single node are performed in fine-grained parallelism (line 10) using the method of [28], where each cut in $E(n)$ is generated using an individual thread, and the selection of the best C cuts that constitutes $P(n)$ are jointly performed by the group of threads in parallel. We skip further details due to the space limit.

Local function checking is interleaved with cut enumeration. At each level, the common cuts of the node pairs whose priority cuts are just computed are generated and inserted into a constant sized buffer (lines 11-12 and 16). When the buffer does not have sufficient space to accept a new batch, an exhaustive simulation is invoked to check all the local functions in terms of the cuts in the buffer and update the equivalence statuses of the proved pairs (lines 13-15). The use of the common cut buffer reduces memory consumption, compared to the method of gathering all the common cuts and performing a single batch of checking in the end.

D. Overall Flow of Simulation-based CEC Engine

Figure 5 depicts the overall flow of the CEC engine which consists of three types of phases: a PO checking phase (`P`), a global function checking phase (`G`), and repeated local function checking phases (`L`).

In the PO checking phase, the CEC engine attempts to prove all or a subset of *simulatable* miter POs in terms of global functions, that is, the primary outputs whose checking can be carried out within a computational budget. The motivation for performing PO checking in the beginning is to maximally reduce the miter so the efforts of checking internal pairs in the removed part of the miter can be saved. A simulatable PO is defined as follows with two parameters k_P, k_p ($k_P > k_p$) that control the budget of checking: if the support sizes of all the POs are no larger than k_P, then all the POs are simulatable; otherwise, a PO is simulatable if its support size is no larger than k_p. The design of two thresholds rather than one is to encourage the CEC engine to prove the miter in a one-shot PO checking if possible.

After initializing equivalence classes by partial random simulation, the global function checking phase verifies simulatable node pairs with support sizes no larger than a threshold k_g, along with collecting CEXs to disprove the nonequivalent pairs and refine the classes.

If the miter cannot be proved by PO and global function checking, the local function checking phases will be conducted repeatedly, where each phase contains three passes of cut generation and checking that attempt to prove all the node pairs as introduced in Section III-C. Since the structure of the miter is modified after reduction at the end of a phase, the cuts generated in the subsequent local checking phase can be different, which provides new chances for verifying the equivalent pairs that failed to be proved in the previous phases. If the miter cannot be further reduced after repeated local function checking, the equivalence of the two circuits is undecided and the reduced miter can be passed to another CEC engine for further proving.

We note that the partial simulator, miter reduction, the EC and CEX manager are all implemented by GPU-parallel algorithms, which ensures the high overall efficiency of the CEC engine.

IV. Experimental Results

The simulation-based CEC engine is implemented in 8,000 lines of CUDA/C++ code on top of the GPU-based logic synthesis tool CULS[2]. We denote it as the "GPU engine" in this section for brevity. The experiments are performed on a Linux machine with Intel Xeon Gold 6326 CPU and NVIDIA RTX A6000 GPU with 48 GB DRAM.

The testcases are selected from the EPFL Combinational Benchmark Suite [29] and the IWLS 2005 Benchmarks [30], covering arithmetic and control designs. To demonstrate the effectiveness of our parallel CEC approach, we enlarge the designs by applying multiple times of the ABC command `double`, which is a common method used in the works on parallel logic synthesis and verification [14], [23], [31]. The two circuits compared by CEC are the original and optimized versions of a benchmark, where the optimized one is generated by executing ABC `resyn2`, an AIG optimization script consisting of several passes of balancing, rewriting and refactoring [32]. Table II shows the statistics of the benchmarks and miters constructed by the pairs of circuits being compared.

In the experiments, we use the following parameter values for the GPU engine: $k_P = 32$, $k_p = k_g = 16$, $k_l = 8$, $C = 8$. The maximum support size of a window after window merging k_s is set as the support size threshold (k_P, k_p, or k_g) of the phase. For instance, in the PO checking phase, $k_s = k_p = 16$ if not all of the POs are simulatable. We adopt the ABC combinational equivalence checker (command `&cec -C 100000`) for further proving a reduced miter if it is undecided (i.e., not empty) after running the GPU engine. A large value is assigned for the maximum number of conflicts during a SAT call (`-C`) since most of the easy-to-prove pairs have already been handled by the GPU engine.

A. Runtime Comparison with ABC and Commercial Checker

We evaluate our approach by comparing it with the ABC checker (command `&cec`) and a commercial verification tool Cadence Conformal LEC (version 19.20). ABC `&cec` is a single-threaded yet high-performance checker based on SAT sweeping. There is no

[2]https://github.com/cuhk-eda/CULS

TABLE II
BENCHMARK STATISTICS AND RUNTIME COMPARISON OF OUR METHOD WITH ABC AND CONFORMAL LEC

Benchmarks	Statistics				ABC `&cec` Runtime (s)	Cfm (16 CPUs) Runtime (s)	Ours (GPU+ABC)				Speed-up	
	#PIs*	#POs*	#Nodes†	Levels†			GPU (s)	Reduced (%)	ABC (s)	Total (s)	vs. ABC	vs. Cfm
hyp_7xd	32768	16384	45881216	24801	7859.26	406002	4616.56	40.2	418.48	5035.04	1.56×	80.64×
log2_10xd	32768	32768	62072832	444	>4 months‡	118392	119633.18	100.0	-	119633.18	88.11×	0.99×
multiplier_10xd	131072	131072	52600832	274	2370.52	3213	159.54	100.0	-	159.54	14.86×	20.14×
sqrt_10xd	131072	65536	44978176	5058	20640.56	30605	52.29	0.7	20623.24	20675.53	1.00×	1.48×
square_10xd	65536	131072	33442816	250	1021.40	2710	144.35	100.0	-	144.35	7.08×	18.77×
voter_10xd	1025024	1024	21862400	70	62610.44	1166	54.20	43.5	35611.63	35665.83	1.76×	0.03×
sin_10xd	24576	25600	10689536	225	2499.28	2081	78.88	100.0	-	78.88	31.68×	26.38×
ac97_ctrl_10xd	2307072	2299904	22685696	12	248.57	1563	97.51	98.9	22.43	119.94	2.07×	13.03×
vga_lcd_5xd	549248	549856	6337536	24	95.82	317	18.51	20.1	81.95	100.46	0.95×	3.16×
Geomean											4.89×	4.88×

* Original/optimized circuit statistics. † AIG miter statistics (excluding contributions of XORs). ‡ Timeout after 122 days. Use 122 days when computing speed-up.
"_$n$xd" in a benchmark name stands for enlarging the benchmark by executing ABC `double` n times.

publicly available parallel checker to our knowledge, so we compare to Conformal which supports parallel checking with up to 16 CPU threads. Although the implementation details of commercial checkers are unknown, it is believed that they perform equivalence checking using a combination of engines [33], and a possible way for multi-threading is to run different engines simultaneously and early stop when an engine finishes.

Table II shows the results of our approach and the runtimes of ABC checker and Conformal. The subcolumn "Reduced (%)" reports the percentages of reduction in miter sizes after executing the GPU engine. It is worth noting that the GPU engine can fully prove (i.e., 100% reduction without the aid of SAT) 4 out of the 9 cases with significant accelerations, including one case (*log2_10xd*) whose runtime is decreased from over 4 months to 1.4 days. Moreover, it solves a large portion of the problem on the cases *hyp_7xd* and *ac97_ctrl_10xd*, as indicated by the fast ABC verification of the reduced miters (subcolumn "ABC (s)") compared to exclusive ABC checking (column "ABC `&cec`"). The GPU engine can only reduce a minor part of the miter on the cases *sqrt_10xd* and *vga_lcd_5xd* but with a short runtime, so the overall solving time is marginally degraded. On average, the combination of GPU engine and ABC checker achieves 4.89× speed-up over the standalone ABC checker.

The combined GPU and ABC checker obtains an averaged 4.88× acceleration and is faster on 7 out of the 9 cases when compared to Conformal with 16 CPU threads. The commercial checker is much more efficient on the case *voter_10xd* and the reason might be that one of the engines in Conformal is well-suited to solving this case.

B. Breakdown Analysis of Simulation-based CEC Engine

Figure 6 shows the runtime percentages of different phases in the GPU engine, which differ across cases. In particular, *log2_10xd* and *sin_10xd* can be directly proved by PO checking (`P`). The rest of the cases (except *sqrt_10xd* that can hardly reduced by the GPU engine) are further analyzed in Figure 7, in which the intermediate miters during the execution of the GPU engine are checked by ABC, and the normalized ABC checking times (over the standalone ABC checking time of the case) are plotted. For instance, the labels PG and PGL refer to the checking times of the miters extracted after the `G` phase and at the end of the GPU flow, respectively. It can be seen that all the three types of phases play important roles in checking some cases, i.e., `P` on *ac97_ctrl_10xd*, `G` on *multiplier_10xd*, *square_10xd*, and `L` on most cases, which demonstrates the effectiveness of the overall design of the GPU CEC engine.

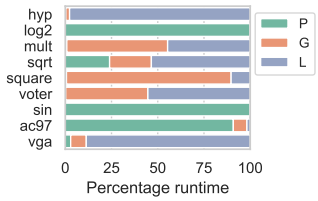

Fig. 6. Runtime breakdown of the simulation-based CEC engine. Benchmark names are abbreviated.

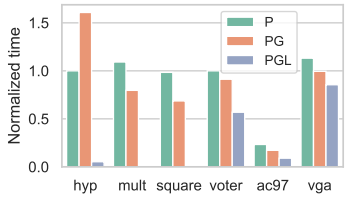

Fig. 7. Time of proving GPU-reduced miter by ABC with different flows, normalized by the time of standalone ABC.

V. DISCUSSION

Our CEC approach is implemented in a minimalist style due to the large development effort required. Here, we discuss a few tweaks that can be applied to further boost checking efficiency.

First, the current integration of the GPU engine and the ABC checker does not implement transferring ECs from GPU to ABC, and the ECs in ABC checking are reinitialized from scratch. With the EC transferring developed, nonequivalent pairs identified by GPU need not be rechecked by ABC, thus reducing the overall runtime.

Second, the GPU flow can be finetuned to be more adaptive, e.g., certain types of passes can be disabled on-the-fly during the repetitive `L` phases if they are found to be ineffective on the case.

Third, some improvements proposed in previous works may also be implemented on GPUs and integrated into our engine, including distance-1 simulation of CEXs [8], reverse simulation [21], and interleaving sweeping with logic rewriting [8], [14].

VI. CONCLUSION

This paper presents a new perspective on checking combinational equivalences using exhaustive simulation and a simulation-based CEC engine implemented on GPUs. The GPU engine is equipped with a fast exhaustive simulator with a high degree of parallelism, a local function checking scheme for proving difficult candidate pairs with restricted computational effort, and other GPU-based managers and algorithms that are crucial to the high overall performance of the engine. Experiments show that the GPU engine can efficiently solve four large cases without the aid of an external checker, and a CEC flow integrating the GPU engine and ABC checker achieves 4.89× and 4.88× speed-up on average over the standalone ABC checker and a commercial verification tool, respectively.

ACKNOWLEDGMENT

The authors acknowledge Yang Sun from CUHK and Jinwei Liu from HKBU for their useful comments and discussions.

REFERENCES

[1] A. Mishchenko, M. Case, R. Brayton, and S. Jang, "Scalable and scalably-verifiable sequential synthesis," in *Proc. ICCAD*, 2008, pp. 234–241.

[2] S. Chatterjee, A. Mishchenko, R. Brayton, X. Wang, and T. Kam, "Reducing structural bias in technology mapping," in *Proc. ICCAD*, 2005, pp. 519–526.

[3] S.-Y. Lee, H. Riener, A. Mishchenko, R. K. Brayton, and G. De Micheli, "A simulation-guided paradigm for logic synthesis and verification," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 41, no. 8, pp. 2573–2586, 2022.

[4] S. Krishnaswamy, H. Ren, N. Modi, and R. Puri, "DeltaSyn: An efficient logic difference optimizer for ECO synthesis," in *Proc. ICCAD*, 2009, pp. 789–796.

[5] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Trans. Comput.*, vol. C-35, no. 8, pp. 677–691, 1986.

[6] A. Kuehlmann and F. Krohm, "Equivalence checking using cuts and heaps," in *Proc. DAC*, 1997, pp. 263–268.

[7] A. Mishchenko, S. Chatterjee, R. Jiang, and R. K. Brayton, "FRAIGs: A unifying representation for logic synthesis and verification," ERL Technical Report, Tech. Rep., 2005.

[8] A. Mishchenko, S. Chatterjee, R. Brayton, and N. Een, "Improvements to combinational equivalence checking," in *Proc. ICCAD*, 2006, pp. 836–843.

[9] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: engineering an efficient SAT solver," in *Proc. DAC*, 2001, pp. 530–535.

[10] N. Eén and N. Sörensson, "An extensible SAT-solver," in *Proc. SAT*, 2003, pp. 502–518.

[11] A. Biere, T. Faller, K. Fazekas, M. Fleury, N. Froleyks, and F. Pollitt, "CaDiCaL 2.0," in *Proc. CAV*, 2024, pp. 133–152.

[12] C. Yu, W. Brown, D. Liu, A. Rossi, and M. Ciesielski, "Formal verification of arithmetic circuits by function extraction," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 35, no. 12, pp. 2131–2142, 2016.

[13] Y. Sun, T. Liu, M. D. Wong, and E. F. Young, "Massively parallel AIG resubstitution," in *Proc. DAC*, 2024, pp. 1–6.

[14] T. Liu and E. F. Young, "Rethinking AIG resynthesis in parallel," in *Proc. DAC*, 2023, pp. 1–6.

[15] R. K. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool," in *Proc. CAV*, 2010.

[16] A. Kuehlmann, V. Paruthi, F. Krohm, and M. Ganai, "Robust Boolean reasoning for equivalence checking and functional property verification," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 21, no. 12, pp. 1377–1394, 2002.

[17] A. Mishchenko and R. Brayton, "SAT-based complete don't-care computation for network optimization," in *Proc. DATE*, 2005, pp. 412–417.

[18] H. Riener, S.-Y. Lee, A. Mishchenko, and G. De Micheli, "Boolean rewriting strikes back: Reconvergence-driven windowing meets resynthesis," in *Proc. ASP-DAC*, 2022, pp. 395–402.

[19] D. Brand, "Verification of large synthesized designs," in *Proc. ICCAD*, 1993, pp. 534–537.

[20] L. Amarú, F. Marranghello, E. Testa, C. Casares, V. Possani, J. Luo, P. Vuillod, A. Mishchenko, and G. De Micheli, "SAT-sweeping enhanced for logic synthesis," in *Proc. DAC*, 2020, pp. 1–6.

[21] H.-T. Zhang, J.-H. R. Jiang, L. Amarú, A. Mishchenko, and R. Brayton, "Deep integration of circuit simulator and SAT solver," in *Proc. DAC*, 2021, pp. 877–882.

[22] H.-T. Zhang, J.-H. R. Jiang, and A. Mishchenko, "A circuit-based SAT solver for logic synthesis," in *Proc. ICCAD*, 2021, pp. 1–6.

[23] V. N. Possani, A. Mishchenko, R. P. Ribas, and A. I. Reis, "Parallel combinational equivalence checking," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 10, pp. 3081–3092, 2019.

[24] D. Chatterjee and V. Bertacco, "EQUIPE: Parallel equivalence checking with GP-GPUs," in *Proc. ICCD*, 2010, pp. 486–493.

[25] NVIDIA, "CUDA C++ programming guide," 2024.

[26] P. Pan and C.-C. Lin, "A new retiming-based technology mapping algorithm for LUT-based FPGAs," in *Proc. FPGA*, 1998, pp. 35–42.

[27] A. Mishchenko, S. Cho, S. Chatterjee, and R. Brayton, "Combinational and sequential mapping with priority cuts," in *Proc. ICCAD*, 2007, pp. 354–361.

[28] T. Liu, L. Chen, X. Li, M. Yuan, and E. F. Young, "FineMap: A fine-grained GPU-parallel LUT mapping engine," in *Proc. ASP-DAC*, 2024, pp. 392–397.

[29] L. Amarú, P.-E. Gaillardon, and G. De Micheli, "The EPFL combinational benchmark suite," in *Proc. IWLS*, 2015.

[30] C. Albrecht, "IWLS 2005 Benchmarks," https://iwls.org/iwls2005/benchmarks.html, 2005.

[31] T. Liu, Y. Sun, L. Chen, X. Li, M. Yuan, and E. F. Young, "A unified parallel framework for LUT mapping and logic optimization," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 44, no. 1, pp. 214–226, 2025.

[32] A. Mishchenko, S. Chatterjee, and R. Brayton, "DAG-aware AIG rewriting: a fresh look at combinational logic synthesis," in *Proc. DAC*, 2006, pp. 532–535.

[33] A. Koelbl, R. Jacoby, H. Jain, and C. Pixley, "Solver technology for system-level to RTL equivalence checking," in *Proc. DATE*, 2009, pp. 196–201.