Massively Parallel AIG Resubstitution

Yang Sun*
CSE Department, CUHK
ysun22@cse.cuhk.edu.hk

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

Martin D.F. Wong
CS Department, HKBU
mdfwong@hkbu.edu.hk

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

ABSTRACT

Resubstitution is a flexible algorithmic framework for circuit restructuring that has been incorporated into many high-effort logic optimization flows. It is thus important to speed up resubstitution in order to obtain high-quality realizations of large-scale designs. This paper proposes a massively parallel AIG resubstitution algorithm targeting GPUs, with effective approaches to addressing cyclic dependencies and restructuring conflicts. Compared with ABC and mockturtle, our algorithm achieves 41.9× and 50.3× acceleration on average without quality degradation. When combining our resubstitution with other GPU algorithms, a GPU-based `resyn2rs` sequence obtains 46.4× speedup over ABC with 0.8% and 5.8% smaller area and delay respectively.

1 INTRODUCTION

Logic optimization is an important stage during the synthesis of digital circuits in which netlists are restructured and simplified in order to improve the area and delay of the realized circuit. One of the most widely used circuit representations during logic optimization is And-Inverter Graph (AIG) [7], and many algorithms for AIG restructuring have been developed over the years which leverage optimization opportunities from different aspects, including rewriting, refactoring [13] and resubstitution [12], to name a few.

The scale of digital designs has become overwhelmingly large in recent years due to the ever-increasing demand of high performance, complex functionality chips as well as the advancement of semiconductor technology nodes. Hence, there emerges a research trend of parallel algorithms for accelerating logic optimization. The works [14] and [8] introduced ways to parallelize AIG rewriting on multi-core CPUs and GPUs respectively, and the paper [10] proposed parallel algorithms for AIG balancing and refactoring. It was reported that 10× to 50× speedup could be achieved on average. For AIG rewriting, the GPU-based algorithm achieves superior acceleration over the multi-core CPU-based counterpart [8], indicating the high performance of GPU for parallel processing applications.

Indeed, compared with multi-core/socket CPU systems, GPUs are able to launch and execute a huge number of threads (e.g., tens of thousands) simultaneously with much less overhead in thread scheduling. This makes GPUs appropriate for applications in which each computation task is typically small, and in contrast the thread scheduling overhead will be significant in the CPU-parallel scenario. Many EDA areas besides logic synthesis have also benefited from such powerfulness of GPUs, ranging from static timing analysis [5], to global routing [9], to design rule checking [6].

In this work, we set our focus on another important algorithm for AIG optimization, namely resubstitution. Resubstitution aims to re-express the function of a gate using some other existing gates (called *divisors*) in the circuit so that the logic of the divisors can be reused and the total number of gates can be reduced. Unlike rewriting and refactoring in which the circuit simplification is done by improving only the local logic of the gate to be optimized, in resubstitution there is no theoretical restriction on the locality of divisors, i.e., a divisor can be arbitrarily far away from the gate, provided that there is sufficient computational budget. Hence, the framework of resubstitution is highly flexible. For instance, the well-known time-costly procedure SAT/BDD sweeping [7] can be regarded as a special case of resubstitution. Besides its flexibility, resubstitution can be enhanced using don't care conditions [11] for achieving better optimization quality, and has been integrated into many high-effort logic synthesis flows [2, 3]. These make resubstitution an important algorithm to be accelerated, which has never been studied from a parallel perspective to our knowledge.

In light of this, we propose a massively parallel algorithm for a commonly used version of AIG resubstitution known as window-based k-resubstitution [12], and its background will be introduced in Section 2.2. The contributions of this paper are:

- Highly parallel procedures for divisor collection, candidate evaluation and network updates.
- A new divisor collection strategy that theoretically prevents cyclic dependency from happening after resubstitution.
- An efficient algorithm for addressing other types of conflicts which ensures race-free parallel network update.

Experiments show that GPU-parallel resubstitution obtains 41.9× and 50.3× speedup compared against the window-based resubstitution implementations in ABC [4] and mockturtle on average, with comparable or better result qualities. Moreover, we integrate GPU resubstitution with other publicly available GPU logic optimization algorithms into a GPU-based `resyn2rs` optimization sequence, which achieves 46.4× acceleration and superior quality over ABC `resyn2rs` with 0.8% smaller area and 5.8% smaller delay.

2 PRELIMINARIES

2.1 Background

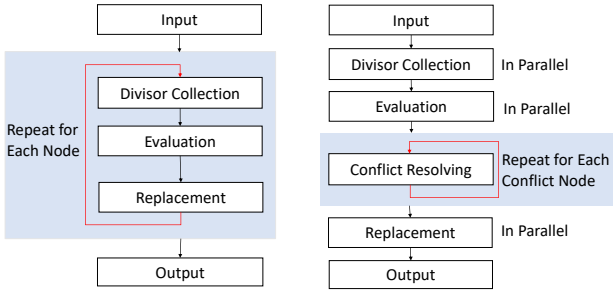
In the context of logic synthesis, a combinational circuit is usually modeled as a *Boolean network*. A Boolean network is a *directed acyclic graph* (DAG) with nodes representing logic functions or gates, and edges corresponding to signals or nets between the nodes. The sources and sinks of the DAG are the *primary inputs* (PIs) and *primary outputs* (POs) of the Boolean network. The *fanins*

*Both authors contributed equally to this research.

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

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

ACM ISBN 979-8-4007-0601-1/24/06
https://doi.org/10.1145/3649329.3655987



(a) Sequential workflow **(b) Our workflow**

Figure 1: Comparison of sequential and our workflow.

and *fanouts* of a node are the direct predecessors and successors of the node respectively. The predecessors of a node are the *transitive fanins* (TFIs) of the node, and all the predecessors form the *transitive fanin cone* of the node. A node's *transitive fanouts* (TFOs) and *transitive fanout cone* can be defined similarly. An *And-Inverter Graph* (AIG) is a Boolean network where each node is an AND gate with exactly two fanins and signals represented by edges can be optionally inverted. The *level of an AIG node* denotes the length of the longest path from a PI to the node, and the *level/delay of an AIG* is the largest level of its POs.

A *cut* C_n of a node n in a Boolean network is a set of nodes such that any path from a PI to n contains a node in the set. A *logic cone* of a node n includes n itself and the intersection of the TFI nodes of n with the TFO nodes of a cut C_n. A *fanout-free cone* (FFC) of a node n is a logic cone of n such that any path from a node in the cone to a PO contains n. All the nodes in the FFC of n are dedicated to driving the logic of n and thus can be removed if n is deleted.

2.2 Window-based k-resubstitution

Window-based k-resubstitution attempts to resynthesize a node under optimization (denoted as *pivot*) using one or more divisors that are collected from a local subgraph near the node (denoted as *window*). All the nodes in the window have a common cut C which is the PIs of the window subgraph. The functions of the pivot and the divisors are expressed in terms of C instead of the PIs of the entire AIG in order to restrict the time complexity of resubstitution.

The parameter k in k-resubstitution stands for the number of new nodes inserted during the resynthesis evaluation of a pivot. For example, if $k = 0$, resubstitution checks whether a divisor is equivalent to the pivot. If $k = 2$, the algorithm finds whether there exists three divisors such that combining them using two AIG nodes (with arbitrary fanin inversion status) yields a node equivalent to the pivot.[1] We denote such a set of $k + 1$ divisors through which a node equivalent to the pivot can be constructed as a *feasible divisor set* D^*. The search for a feasible divisor set is usually done by an exhaustive enumeration of all the candidate divisors, leading to $O(m^{k+1})$ complexity for checking one pivot where m is the number of candidate divisors. Besides, a feasible k-resubstitution is only beneficial if the number of nodes deleted (i.e., the number of FFC nodes of the pivot) is larger than k in order to achieve area reduction. Due to these two reasons, k is usually bounded by 2 or 3 in practice.

[1]The equivalences here are up to complementation.

3 PARALLEL AIG RESUBSTITUTION

In this section, we introduce our parallel k-resubstitution. First, we give an overview of the framework. We then describe our divisor collection strategy tailored for avoiding cyclic dependencies in parallel settings, and details in candidate divisor evaluation. Finally, we present an effective and fast conflict resolution method for detecting and addressing potential conflicts brought by parallelization.

3.1 Overview of Parallel Resubstitution

Figure 1 shows the sequential and the proposed parallel workflow. In sequential resubstitution, divisor collection, evaluation and replacement are repeated sequentially for each node in the AIG, as shown in Figure 1a. In our framework, these three procedures are parallelized in which each thread performs the corresponding computational task of a single node.

Divisor collection and evaluation are the most time-consuming procedures in classical resubstitution algorithms [4]. In divisor collection, every node constructs a window and only cares about the local function inside the window. Such property enables massive parallelization because every thread can focus on one node and operate locally without the need for inter-thread communication.

However, collecting divisors without careful consideration might result in cyclic dependency, which has to be resolved with extra effort. To avoid that, we propose a cycle-free divisor collection algorithm where cyclic dependency is proven to be impossible. The explanation of such dependency and the algorithm details will be given in Section 3.2.

The evaluation procedure for searching feasible divisor sets is described in Section 3.3. If some divisors are evaluated as feasible, they can be applied to replace the original AIG. However, there are three conditions under which the replacements might be conflicting, as will be shown in Section 3.4.1. To resolve the conflict issue, we first apply a parallel conflict identification algorithm to accept those replacements that do not conflict with any other replacement. An efficient method is then applied to resolve the remaining conflicts in sequence. Details will be given in Section 3.4.2. With all conflicts having been resolved, new structures can be updated in parallel without any data race.

3.2 Divisor Collection for Cycle-free Parallel Resubstitution

During resubstitution, given a cut C_n of a pivot node n, a window W_n of n is constructed as a restricted TFO subgraph of C_n such that C_n is a cut of all the nodes in W_n. The constraint can be a maximum window size, or a maximum number of levels beyond the pivot. Figure 2 shows an example where the windows only contain nodes with a level at most one higher than the corresponding pivot.

Traditionally, given a window W_n, the divisors of the pivot n are essentially the nodes in $V(W_n) \cup C_n$ but with two kinds of nodes excluded. First, any FFC node of n in W_n is not considered as a divisor, since it can be deleted if there is a successful resubstitution. Second, any TFO of n cannot be a divisor, otherwise there may be a cyclic dependency involving the divisor and the pivot after resubstitution and the resulting AIG will be invalid, which is illustrated by Figure 3. These two exceptions during divisor collection ensure the effectiveness and correctness of sequential resubstitution.

Algorithm 1 Cycle-Free Divisor Collection (per thread)

Input: Pivot n, cut C, maximum number of candidate divisors M_{max}
Output: Candidate divisors D

1: $D \leftarrow C$ ▷ collect cut nodes first
2: $TFI \leftarrow TFI(n, C)$, $FFC \leftarrow FFC(n, C)$
3: **for each** $d \in TFI$ **do**
4: **if** $d \notin FFC$ **then**
5: $D \leftarrow D \cup d$
6: **for each** $d \in D$ **do** ▷ iteratively expands the window
7: **for each** $o \in d.fanouts$ **do**
8: **if** $o \in C$ or $o \in FFC$ **then continue**
9: **if** $o.level > n.level$ **then continue**
10: **if** $o.level = n.level$ and $o.id > n.id$ **then continue**
11: **if** $o \notin D$ and $o.fanins \subset D$ **then**
12: $D \leftarrow D \cup o$
13: **if** $|D| = M_{max}$ **then return**

In the parallel scenario, however, such formulation of divisor collection fails to prevent the aforementioned cyclic dependency from happening. To see why a cycle may still exist, consider the example shown in Figure 2. Note that p_2 is a divisor of pivot p_1, and p_1 is also a divisor of pivot p_2. If a feasible divisor set of p_1 contains p_2 and vice versa, there will be a cycle of length 2 after resubstitution in which p_1 and p_2 depend on each other. It is not difficult to see that longer cyclic dependencies may also happen, e.g., $n_1 \leftarrow n_2 \leftarrow n_3 \leftarrow n_1$ for a cycle of length 3.

To address this issue, we introduce a new constraint during divisor collection for parallel resubstitution. The new approach can theoretically guarantee that there is no cycle after parallel resubstitution, which is formalized as Proposition 1.

Proposition 1. *If the collected divisor sets of all the nodes in an AIG contain only (1) nodes with a level smaller than the corresponding pivot, or (2) nodes with the same level of the corresponding pivot but a smaller node id, then there will be no cyclic dependency after parallel resubstitution.*

Proof. Consider a directed dependency graph $G = (V, E)$ that captures the dependency between a node and its divisors. In G, the node set V is identical to the node set of the AIG, and for each node n, we add an edge (d, n) to E for every divisor d in the collected candidate divisor set of n. It then suffices to show that G is acyclic in order to prove the original statement.

For any edge $e = (n, m) \in E$, we define $\Delta_l(e) = m.level - n.level$ and $\Delta_i(e) = m.id - n.id$. Suppose that there exists a cycle $\Pi = (e_1, e_2, \ldots, e_k) = ((n_1, n_2), (n_2, n_3), \ldots, (n_k, n_1))$ in G. It is clear that $\sum_{e\in\Pi} \Delta_l(e) = 0$ and $\sum_{e\in\Pi} \Delta_i(e) = 0$. Since the divisor sets only contain nodes with a level no greater than the corresponding pivot, we have $\Delta_l(e) \geq 0$ for any edge $e \in E$, and hence $\Delta_l(e) = 0$ for any edge $e \in \Pi$, i.e., all the nodes involved in the cycle Π have the same level. This implies that $\Delta_i(e) > 0$ for any edge $e \in \Pi$ because a divisor whose level is the same as its pivot needs to have a smaller id. However, this contradicts $\sum_{e\in\Pi} \Delta_i(e) = 0$, and we conclude that G is acyclic. □

Algorithm 1 elaborates our cycle-free divisor collection. The candidate divisor set D is initialized with the cut nodes (line 1) and the non-FFC nodes in the TFI cone of the pivot bounded by cut C (line 3-5). The window then expands towards the PO by adding new divisors that satisfy the constraint stated in Proposition 1 (line 9-10). To ensure that a divisor does not functionally depend on a node not in C, it is only accepted if both of its two fanins have already been collected (line 11-12). This condition also excludes all TFOs of the pivot n, because n is never collected in D. Window expansion stops when there is no new node that can be collected as a divisor or the limit of the number of divisors is reached. Note that we do not explicitly maintain a data structure for the window, since the window can be represented as $D \cup FFC \setminus C$ (FFC and C are constant sets) at any time during the construction of D.

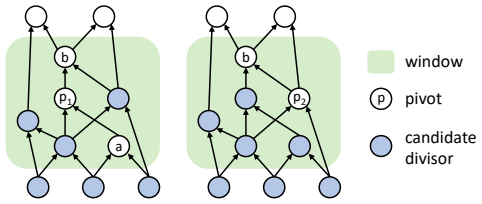


Figure 2: An example of windows and divisors. "a" and "b" indicate the two kinds of nodes that should be excluded from divisor collection respectively, as introduced in Section 3.2.

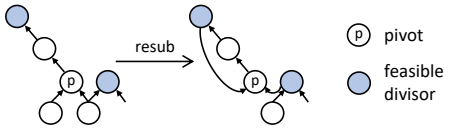


Figure 3: A TFO divisor may cause a cycle after resubstitution.

3.3 Candidate Divisor Evaluation

Once the candidate divisors are collected, we need to find out if there exist some combinations of divisors whose function is equal to the pivot. An intuitive way is to exhaustively try all the combinations of the candidate divisors. However, some candidate divisors can be trivially identified as infeasible, and there is no need to spend much effort on them. So, the search space can be greatly reduced if such infeasible divisors are filtered out first.

3.3.1 Candidate Filtering. We take 1-resub as an example to explain candidate filtering. In 1-resub, two nodes might be connected by an AND or OR gate, i.e., $a \wedge b$ or $a \vee b$.[2] For a candidate divisor a, it is possible that $f \neq a \wedge b, \forall b \in D$. For example, suppose the function of a pivot f is expressed in truth table as $f = 0011$, and there are five candidates whose functions are $a = 0010$, $b = 0001$, $c = 1011$, $d = 0111$ and $e = 0101$. A condition that is able to identify an infeasible candidate using AND gate is specified in Proposition 2.

Proposition 2. *Let f be the function of the pivot node to be replaced by a k-input AND gate, and f_i is the function of the i-th input d_i. Then, the resubstitution $f = \bigwedge_{i=1}^{k} f_i$ is valid only if $f \rightarrow f_i, \forall i$.*

Proof. If $f \not\rightarrow f_{i'}$, then there exists a bit j such that $f_j = 1$ and $(f_{i'})_j = 0$. Hence $f_j \neq \bigwedge_{i=1}^{k} (f_i)_j$ and $f \neq \bigwedge_{i=1}^{k} f_i$. □

[2]For brevity, the complementation of a node/function is ignored in this section, but our discussion can be easily applied to such cases by replacing variables with literals.

Algorithm 2 Two-stage Conflict Handling

Input: Pivot set P, feasible divisors D_p^* and FFC_p of each pivot

```
1: function ParCheck(pivot p, feasible divisors D_p^*, FFC_p)
2:     if ChangeOfFFC(FFC_p) then return
3:     if AncestorDeletion(p) then return
4:     if DivisorDeletion(D_p^*) then return
5:     markDeleted(p, FFC_p)                    ▷ no conflict, accept resub
6: end function
7:
8: function SeqCheck(pivot p, feasible divisors D_p^*, FFC_p)
9:     for each d ∈ D_p^* do
10:        if isDeleted(d) then return          ▷ conflict, reject resub
11:    FFC_p ← updateFFC(FFC_p)
12:    if |FFC_p| > k then
13:        markDeleted(p, FFC_p)
14: end function
15:
16: for each p ∈ P in parallel do               ▷ parallel stage
17:    ParCheck(p, D_p^*, FFC_p)
18: for each p ∈ P do                           ▷ sequential stage
19:    if not isDeleted(p) then
20:        SeqCheck(p, D_p^*, FFC_p)
21: ParallelReplacement()
```

A similar condition exists in the OR case: $f = \bigvee_{i=1}^{k} f_i$ is valid only if $f_i \rightarrow f, \forall i$. In our example, as neither $f \rightarrow e$ nor $e \rightarrow f$ holds, we conclude that any 1-resub combination involving e is not equal to f, and thus e can be filtered out in 1-resub.

3.3.2 Candidate Evaluation. Filtering-enhanced candidate evaluation is specified as follows. In AND-resub, we can maintain a set $D_A = \{d_i \mid f \rightarrow d_i,\ d_i \in D\}$, so only the combinations such as $f = a \wedge b, a, b \in D_A$ need to be checked. A similar method can be applied to OR-resub by maintaining a set $D_O = \{d_i \mid d_i \rightarrow f,\ d_i \in D\}$ and only those combinations of $f = a \vee b, a, b \in D_O$ are checked.

This technique is also applicable to 2-resub and 3-resub. For 2-resub, there are four cases:

$$f = \begin{cases} a \vee b \vee c & \text{only if } a \rightarrow f, b \rightarrow f, c \rightarrow f; \\ a \wedge b \wedge c & \text{only if } f \rightarrow a, f \rightarrow b, f \rightarrow c; \\ a \vee (b \wedge c) & \text{only if } a \rightarrow f, (b \wedge c) \rightarrow f; \\ a \wedge (b \vee c) & \text{only if } f \rightarrow a, f \rightarrow (b \vee c). \end{cases}$$

The first two cases are essentially the same as 1-resub, except that three divisors are chosen from D_O or D_A. For the last two cases, we can apply the technique recursively, e.g., maintain a set $D_{O2} = \{d_i \wedge d_j \mid d_i \wedge d_j \rightarrow f, d_i, d_j \in D\}$.

3.4 Resolving Conflicts

After candidate evaluation, beneficial replacements with positive gains in area are found for some pivot nodes. However, not all replacements can be applied to update the AIG due to some potential conflicts which will be introduced in Section 3.4.1. To resolve these conflicts, we design a two-stage checking algorithm, including a parallel and a sequential stage. Details can be found in Section 3.4.2.

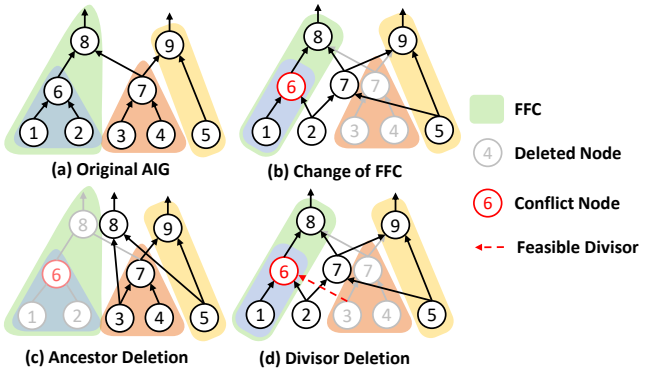


Figure 4: Example of conflict replacement.

3.4.1 Conflict Cases. In resubstitution, a pivot node can be replaced by the nodes in its feasible divisor set, and the pivot together with its FFC will be deleted. However, on some occasions, two replacements can be incompatible, and at least one of them needs to be rejected. There are three cases where replacements might be invalid, which are illustrated in Figure 4.

Case 1: Change of FFC. If a pivot node p has an FFC node that is used to replace another node a, we cannot replace p and a simultaneously. For example, in Figure 4b, the replacement of node 7 needs to use node 2, which is in the FFC of node 6. If we replace node 6 at the same time and delete its FFC, node 2 will be removed and cannot be used to replace node 7.

Case 2: Ancestor Deletion. A pivot node p and its ancestor a cannot be replaced at the same time if p is in the FFC of a. The reason is that by replacing a, p will be deleted, so its replacement would be meaningless. For instance, in Figure 4c, the replacement of node 6 is invalid if node 8 is also replaced.

Case 3: Divisor Deletion. If one of the feasible divisors of a pivot node is deleted, the replacement will be invalid. For example, in Figure 4d, the replacement of node 7 will delete node 3, which is a feasible divisor of node 6, so we cannot perform the replacement of both node 6 and 7 at the same time.

The detection of the three cases is efficient. For case 1, we can check whether an FFC node of a pivot node p is a feasible divisor of another node q. If so, the replacement of p is incompatible with that of q. To detect case 2 of ancestor deletion, we will trace back from a pivot node p iteratively to its ancestor nodes as long as p is in the ancestor's FFC. If an ancestor node q has feasible divisors, the replacement of p is in conflict with that of q. For case 3 of divisor deletion, there will be conflict when a pivot's feasible divisor is deleted. A divisor node might be deleted if one of its ancestors is replaceable, which is similar to the detection of case 2. We will thus traverse the feasible divisors of a pivot node, and detect ancestor deletion. If any feasible divisor fails in the detection, a divisor deletion is found. Note that we do not need to check for cyclic conflicts because of Proposition 1.

3.4.2 Details of Conflict Resolving. Conflict resolving is divided into two stages as shown in Algorithm 2. First, every pivot node with a feasible replacement is checked in parallel. In this stage, we

Table 1: Comparison of window-based k-resubstitution (k=2).

Benchmarks	Statistics		ABC `resub`			mockturtle `aig_resub`			GPU `resub`		
	#Nodes	Levels	#Nodes	Levels	Time (s)	#Nodes	Levels	Time (s)	#Nodes	Levels	Time (s)
sixteen	16216836	140	16143196	140	235.9	16173172	140	275.3	16140545	140	5.0
twenty	20732893	162	20648660	162	317.8	20680795	162	370.2	20648313	162	6.3
twentythree	23339737	176	23249547	176	357.8	23283697	176	431.7	23249102	176	7.6
div_10xd	58620928	4372	46350336	4372	798.0	46245888	4402	1537.8	46311424	4405	22.6
hyp_8xd	54869760	24801	53232640	24801	778.2	53149696	24804	448.6	54393600	24802	25.3
mem_ctrl_10xd	47960064	114	47481856	114	631.1	47731712	114	748.9	47640576	114	10.5
log2_10xd	32829440	444	32001024	433	554.9	32043008	435	456.9	31578112	423	11.4
multiplier_10xd	27711488	274	26878976	273	394.1	26966016	273	441.1	26624000	272	8.6
sqrt_10xd	25208832	5058	21885952	5058	325.3	21013504	5900	438.6	21151744	5990	11.0
square_10xd	18927616	250	17263616	250	264.9	18396160	250	238.4	17537024	250	5.9
voter_10xd	14088192	70	9511936	65	161.5	10733568	73	136.0	9764864	69	3.9
sin_10xd	5545984	225	5372928	225	82.3	5428224	227	87.4	5354496	223	2.1
ac97_ctrl_10xd	14610432	12	14294016	12	176.1	13800448	12	862.0	13766656	12	4.4
vga_lcd_5xd	4054752	24	3809056	24	133.8	3809984	24	165.8	3810368	24	3.6
Geomean Ratio			1.000	1.000	41.9	1.009	1.021	50.3	0.998	1.014	1.0

"_*n*xd" means that the benchmark is generated by enlarging the original one using ABC `double` *n* times.

Table 2: Comparison of `resyn2rs` sequence.

Benchmarks	ABC `resyn2rs`			GPU `resyn2rs`*		
	#Nodes	Levels	Time (s)	#Nodes	Levels	Time (s)
sixteen	11970378	99	8530.2	11781381	64	63.2
twenty	15309087	86	9763.8	15106639	65	81.2
twentythree	17160203	94	11809.5	16942499	68	91.8
div_10xd	41717760	4370	12581.6	41646619	4413	381.2
hyp_8xd	52361984	24792	25026.0	53181303	24671	775.2
mem_ctrl_10xd	44986368	112	13269.5	43365698	91	403.1
log2_10xd	29893632	376	12815.1	29998592	357	339.3
multiplier_10xd	24922112	262	8505.2	24968192	262	184.2
sqrt_10xd	19584000	4968	6979.8	18688000	5927	206.6
square_10xd	16272384	248	5409.1	16237303	246	108.0
voter_10xd	8138752	57	3026.8	8220679	61	48.7
sin_10xd	5141504	175	2116.4	5143988	165	78.4
ac97_ctrl_10xd	10604544	9	2321.0	10526720	9	65.8
vga_lcd_5xd	2906272	18	2302.2	2903809	24	135.5
Geomean Ratio	1.000	1.000	46.4	0.992	0.942	1.0

* `-K 10` for the last two `rs` commands.

first assume that all replacements are accepted. Under this assumption, if a pivot does not have any conflict type introduced above (line 2-4), we can confirm its replacement and mark its FFC (including the pivot itself) as deleted (line 5). Conversely, if a pivot node is not marked as deleted after the parallel checking, there might be some conflicts with it and sequential checking is therefore needed.

In the sequential stage, we check the remaining pivot nodes (line 18-19) one by one. The sequential checking is done greedily: we accept a replacement as long as it is feasible, i.e., all of the pivot's feasible divisors exist, and reject a replacement otherwise (line 9-10). If there is no deleted feasible divisor, we could consider accepting the replacement, and the pivot's FFC needs to be updated (line 11) because it might shrink as shown in Figure 4b. After that, if the replacement is still considered beneficial (line 12), we finally confirm the acceptance of the resubstitution and delete the updated FFC.

After every pivot having been checked, the status of the nodes in the AIG are determined: a node can be deleted, replaced, or unchanged. One GPU thread is then assigned to a node to update its structure. Since all the conflicts have been resolved, the network update can be conducted in parallel and no data race would happen.

4 EXPERIMENTAL RESULTS

The proposed parallel window-based *k*-resubstitution is implemented using CUDA C/C++, and the experiments are conducted on a server with Intel Xeon Silver 4114 CPU and NVIDIA GeForce RTX 3090 GPU with 24GB DRAM. The benchmarks are selected from the EPFL Combinational Benchmark Suite [1] including three MtM random Boolean functions, with two control circuits from the IWLS 2005 Benchmarks[3]. The small cases are enlarged in order to illustrate the real performance of massive parallelization, following the approach in [10]. Table 1 shows the statistics of the benchmarks.

We set the maximum number of new nodes during resubstitution as 2 (i.e., $k = 2$) in all the experiments unless otherwise specified. All the results generated by our program passed combinational equivalence checking.

4.1 Comparison of k-resubstitution

We compare our algorithm with two sequential window-based *k*-resubstitution implemented in the logic synthesis tool ABC [4] and mockturtle[4] respectively. As far as we know, there is no known parallel implementation currently.

As shown in Table 1, GPU resubstitution on average achieves 41.9× and 50.3× acceleration over ABC and mockturtle respectively. For the quality of the results, GPU resubstitution obtains the smallest area and the second smallest delay. This indicates that our parallel algorithm is able to significantly speed up resubstitution while preserving high optimization qualities.

4.2 Comparison of Optimization Sequence

In order to simplify a circuit substantially, it is common in practice to apply a sequence of different optimization algorithms rather than performing a particular algorithm repeatedly. Hence, we test our algorithm in the context of an ABC sequence `resyn2rs`, specified as:

```
b; rs -K 6; rw; rs -K 6 -N 2; rf; rs -K 8;
b; rs -K 8 -N 2; rw; rs -K 10; rw -z; rs -K 10 -N 2;
b; rs -K 12; rf -z; rs -K 12 -N 2; rw -z; b,
```

[3]https://iwls.org/iwls2005/benchmarks.html
[4]https://github.com/lsils/mockturtle

Table 3: Comparison of different divisor collection strategies.

Method	Norm. #Nodes	Norm. Time
ours	1.000	1.0
full+resolve_cycle	0.997	2.3
smaller_level	1.010	1.0

where `b`, `rw`, `rf`, `rs` denote AIG balancing, rewriting, refactoring and resubstitution. The option `-K` and `-N` of `rs` stand for the maximum cut size of the window, and the maximum number of inserted nodes[5] respectively. The GPU-parallel versions of `b`, `rw` and `rf` are available in the logic synthesis tool CULS[6], so we integrate them with GPU resubstitution and obtain a GPU-parallel `resyn2rs`.

The results comparing GPU versus ABC `resyn2rs` are shown in Table 2. Due to insufficient GPU memory, we set `-K 10` (originally `-K 12`) for the last two `rs` in GPU `resyn2rs`. Under such adjustment, GPU `resyn2rs` still shows superior performance over the ABC counterpart by achieving 0.8% smaller area and 5.8% smaller delay with 46.4× acceleration, which demonstrates the effectiveness of the proposed algorithm in real-world logic optimization scenarios.

4.3 Divisor Collection Strategies

As introduced in Section 3.2, parallel resubstitution ensures cycle-freeness by introducing an additional constraint during divisor collection. In other words, some candidate divisors are not included in our settings compared with the traditional collection method. To investigate whether this affects the performance of resubstitution, we compare our divisor collection strategy (specified in Proposition 1) against two variants:

Variant 1 (full+resolve_cycle): the traditional divisor collection method is used. Since this may introduce cyclic dependencies, we additionally perform cycle detection and resolving for each node on-the-fly during the sequential conflict checking stage (Section 3.4).

Variant 2 (smaller_level): only divisors with a smaller level than that of the pivot will be considered. This is even stricter than our constraint and leads to a smaller number of collected divisors. Cycle-freeness can be guaranteed.

The performances of GPU resubstitution with these strategies are shown in Table 3. The traditional divisor collection strategy slightly reduces the area of the results by 0.3% compared to our method, but resolving cycles induces a huge runtime overhead. On the other hand, further restricting divisor collection leads to a noticeable drop in optimization quality without any runtime benefit. To conclude, our divisor collection strategy is a key component that ensures efficient parallel resubstitution without sacrificing quality.

4.4 Scaling Experiments

Figure 5 shows the speedup of parallel resubstitution over ABC on benchmarks enlarged to different sizes. Notably, even on the original AIGs without enlargement (leftmost point of each curve) our algorithm runs faster than ABC for all cases shown except *sqrt*. In general, the acceleration ratio increases with the increasing circuit size, but seems to saturate when there are more than 10^7 nodes which could be due to limited GPU computational resources.

[5]In ABC, `-N` refers to the same concept as k in our notation.
[6]https://github.com/cuhk-eda/CULS

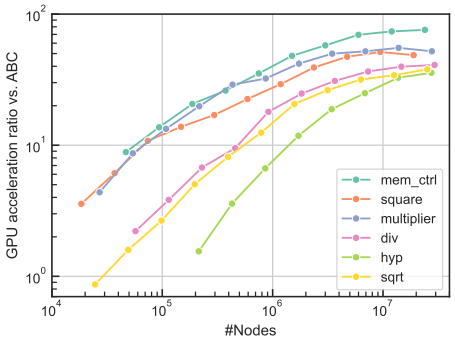


Figure 5: Result of scaling experiments.

5 CONCLUSION

In this paper, we propose an efficient GPU-parallel framework for window-based k-resubstitution. Specifically, the procedures of divisor collection and evaluation for all AIG nodes are done in parallel, and a new strategy for divisor collection is employed which theoretically ensures cycle-freeness after resubstitution. Moreover, we design an algorithm for detecting and resolving conflicts, which prevents potential data races from happening during parallel replacement. On average, GPU resubstitution achieves 41.9× and 50.3× acceleration over ABC and mockturtle on large AIG benchmarks, with comparable or better qualities. We further combine our algorithm with other GPU logic optimization algorithms obtaining a GPU-based `resyn2rs`, which speeds up the ABC counterpart by 46.4× with superior optimization quality.

REFERENCES

[1] Luca Amarú, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. 2015. The EPFL combinational benchmark suite. In *Proc. IWLS*.
[2] Luca Amarú, Vinicius Possani, Eleonora Testa, Felipe Marranghello, Christopher Casares, Jiong Luo, Patrick Vuillod, Alan Mishchenko, and Giovanni De Micheli. 2021. LUT-based optimization for ASIC design flow. In *Proc. DAC*. 871–876.
[3] Luca Amarú, Mathias Soeken, Patrick Vuillod, Jiong Luo, Alan Mishchenko, Janet Olson, Robert Brayton, and Giovanni De Micheli. 2018. Improvements to Boolean resynthesis. In *Proc. DATE*. 755–760.
[4] Robert K. Brayton and Alan Mishchenko. 2010. ABC: An Academic Industrial-Strength Verification Tool. In *Proc. CAV*.
[5] Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. 2020. GPU-accelerated static timing analysis. In *Proc. ICCAD*. 1–9.
[6] Zhuolun He, Yuzhe Ma, and Bei Yu. 2022. X-Check: GPU-accelerated design rule checking via parallel sweepline algorithms. In *Proc. ICCAD*. 1–9.
[7] A. Kuehlmann, V. Paruthi, F. Krohm, and M.K. Ganai. 2002. Robust Boolean reasoning for equivalence checking and functional property verification. *IEEE TCAD* 21, 12 (2002), 1377–1394.
[8] Shiju Lin, Jinwei Liu, Tianji Liu, Martin D.F. Wong, and Evangeline F.Y. Young. 2022. NovelRewrite: node-Level parallel AIG rewriting. In *Proc. DAC*.
[9] Shiju Lin, Jinwei Liu, Evangeline F. Y. Young, and Martin D. F. Wong. 2023. GAMER: GPU-accelerated maze routing. *IEEE TCAD* 42, 2 (2023), 583–593.
[10] Tianji Liu and Evangeline F.Y. Young. 2023. Rethinking AIG resynthesis in parallel. In *Proc. DAC*. 1–6.
[11] Alan Mishchenko, Robert Brayton, Jie-Hong R Jiang, and Stephen Jang. 2011. Scalable don't-care-based logic optimization and resynthesis. *ACM TRETS* 4, 4 (2011), 1–23.
[12] Alan Mishchenko and Robert K. Brayton. 2006. Scalable logic synthesis using a simple circuit structure. In *Proc. IWLS*.
[13] Alan Mishchenko, Satrajit Chatterjee, and Robert Brayton. 2006. DAG-aware AIG rewriting: a fresh look at combinational logic synthesis. In *Proc. DAC*. 532–535.
[14] Vinicius Possani, Yi-Shan Lu, Alan Mishchenko, Keshav Pingali, Renato Ribas, and Andre Reis. 2018. Unlocking Fine-Grain Parallelism for AIG Rewriting. In *Proc. ICCAD*. 1–8.