Rethinking AIG Resynthesis in Parallel

Tianji Liu Evangeline F.Y. Young
Department of Computer Science and Engineering, The Chinese University of Hong Kong
tjliu@cse.cuhk.edu.hk, fyyoung@cse.cuhk.edu.hk

Abstract—**The efficiency issue of logic optimization becomes critical as the scale of VLSI designs grows. Since various algorithms are interleaved during optimization to ensure quality, it is necessary to accelerate those commonly used algorithms for obtaining substantial total speed-up. This paper proposes novel parallel algorithms for AIG refactoring and AND-balancing. Equipped with delicately designed parallel-friendly, data-race-free frameworks and GPU data structures, our algorithms obtain significant speed-up and enable the `resyn2` sequence to be fully GPU-parallelized when combined with GPU rewriting. Experiments show that on large AIGs, we achieve average accelerations up to $45.9\times$ over ABC with comparable or better qualities.**

I. Introduction

Logic optimization serves as a crucial part in logic synthesis and VLSI design flows. In technology-independent logic optimization, And-Inverter Graph (AIG) [1] is one of the most commonly used logic representations because of its simplicity and flexibility. It is equipped with a series of well-developed optimization algorithms including but not limited to AND-balancing [2], [3], rewriting [4], refactoring [4] and resubstitution [5], implemented in the academic logic synthesis tool ABC [6]. The algorithms are usually assembled into sequences which define the order they are applied to a circuit. For example, a commonly used sequence is `resyn2` (consisting of AND-balancing, rewriting and refactoring), which also serves as subroutines in other logic synthesis algorithms such as structural choice computation [7] for technology mapping.

With the ever-growing scale of VLSI designs, logic optimization has become a time-costly process. For instance, it requires nearly an hour for running the ABC `resyn2` sequence on a ten-million-node AIG. There are works targeting at accelerating logic optimization through parallelization, but to the best of our knowledge, they all focus on rewriting [8], [9]. From a practical viewpoint, this is still inadequate. Since rewriting only changes local structures, applying rewriting exclusively leads to bad quality of results. In contrast, other algorithms such as refactoring and AND-balancing optimize much larger structures or perform global restructuring. Hence, to ensure quality, different algorithms need to be applied in an interleaving way, following an optimization sequence. Apparently, the runtime of an optimization sequence cannot be significantly reduced if only rewriting is accelerated. This motivates us to look into the parallelization of other important algorithms besides rewriting.

Recently, GPUs have been extensively utilized in accelerating EDA algorithms, e.g., global placement [10], maze routing [11], as well as AIG rewriting in logic optimization [9]. Thanks to its specifically designed architecture for massive parallelism, GPUs can obtain very high data processing throughput by simultaneously executing tens of thousands of threads scheduled by the backend driver, and it is often observed that GPU-accelerated EDA algorithms can be much faster than multi-threaded CPU implementations [9].

In light of these, we propose GPU-parallel algorithms for AIG refactoring and AND-balancing. In the following, we will refer to AND-balancing as *balancing* for brevity. Although the replacement step of GPU rewriting [9] can be adopted in refactoring, the efficiency will be significantly affected, since this step is not parallelized in [9] due to data races. Instead, we propose a novel framework for GPU refactoring that enables *parallel replacement without data race*, and achieves high acceleration over ABC with even better quality of results. For balancing, we reformulate the recursive and hard-to-parallelize ABC implementation into a form that is friendly to parallelization. We describe the algorithms without involving details related to the underlying programming model and interface, so theoretically they are applicable to any kind of massively parallel processor. Our technical contributions are:

- Developing auxiliary GPU data structure and algorithms serving as important components of GPU refactoring and balancing as well as the infrastructure of a GPU logic optimization tool.
- Implementing the proposed parallel refactoring and balancing on CUDA-enabled GPUs, achieving acceleration ratios of $42.7\times$ and $14.8\times$ respectively over ABC on large AIG benchmarks.
- Creating a full-GPU accelerated `resyn2` sequence by integrating GPU rewriting [9] with our proposed algorithms, achieving $45.9\times$ acceleration over ABC without quality degradation.

II. Preliminaries

A. Backgrounds

A *Boolean network* is a directed acyclic graph (DAG) in which each node corresponds to a logic function and edges represent input/output signals of nodes. The predecessors and successors of a node are called the *transitive fanins* and *transitive fanouts* of the node respectively. The direct predecessors and successors of a node are called *fanins* and *fanouts* respectively. The *primary inputs* (PIs) and *primary outputs* (POs) of a Boolean network are the sources and sinks of the DAG respectively. The function at each node, taking its fanin signals as inputs, is denoted as the *local function* of the node. An *And-Inverter Graph* (AIG) is a Boolean network in which all nodes are two-input AND gates with optionally complemented fanin signals. In an AIG, the *delay of a node* is the length of the longest path from any PI to this node, computed recursively as the maximum delay of its fanin nodes plus one (the signal delay at this node). The *delay/level of an AIG* is the maximum delay of all POs.

For a node n in an AIG, a *cut* of n is a set of nodes in the AIG such that any path from a PI to n passes through at least one node in the set. The *logic cone* associated with a cut of n includes n, and the intersection of all the transitive fanins of n and all the transitive fanouts of the cut nodes.

Structural hashing [5] is an implementation technique that ensures the uniqueness of an AIG node, relying on a hash-table. Whenever a new AND node is to be created, structural hashing checks whether a node with the same key (two fanins with complement status) exists. If so, the existing node will be reused for the new one.

B. AIG Optimization Algorithms

a) Sequential algorithms: Although rewriting and refactoring [4] both optimize local subgraph structures, there exist significant differences between them. In refactoring, for each AND node visited in a topological order, one large cut is computed for the node. The

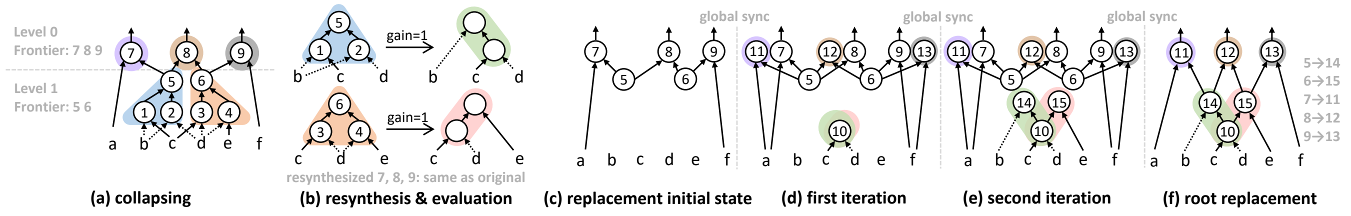

Fig. 1: A full example for GPU refactoring.

local function taking the cut nodes as inputs is then resynthesized following a standard factoring [12] procedure. The new subgraph will be replaced into the AIG if the number of AIG nodes is reduced or not increased. For rewriting, the candidate subgraphs are instead retrieved from a pre-computed library. Besides, refactoring can handle much larger subgraphs (cuts) than rewriting does.

For balancing [2], it aims to reduce the delay of an AIG. The core step of balancing is essentially re-combining some internal nodes in a certain order, thus enabling a more global structural change compared to rewriting or refactoring. The implementation detail of balancing in ABC [6] is provided in Section IV-A.

b) GPU-parallel rewriting: In GPU rewrite [9], the best cone for all nodes is computed and *inserted* into the original AIG (if better than the original cone) in parallel, resulting in functionally equivalent pairs of cones rooted at some nodes. Then, for each such pair of cones, the better one is kept, by an on-the-fly re-evaluation process and the worse one will be deleted. This step is done sequentially to avoid data race, but from our observations, this becomes the bottleneck of the algorithm.

III. GPU-parallel Refactoring

As mentioned in Section II-B, in GPU rewrite [9], the replacement step is performed sequentially. If replacement is parallelized, one must carefully deal with the data race issues such as multiple threads attempting to delete the same node. Sequential replacement does not significantly affect the efficiency of GPU rewrite, but in refactoring, the replacement step is computationally more expensive due to larger cut and cone sizes. Experiments show that the runtime of the sequential part is 60% longer than that of GPU rewrite, as shown in Table I. Hence, we need a new parallel replacement method in order to obtain significant speedup for the refactoring algorithm.

A. The Big Picture of a Novel Parallel Framework

We begin by introducing fanout-free cones and maximum fanout-free cones, which play important roles in our refactoring framework.

Definition 1. A *fanout-free cone* (FFC) of a node n is a logic cone of n (associated with any cut of n), such that for each node n' in the cone, any path from n' to any PO passes through n. The *maximum fanout-free cone* (MFFC) of a node n is the largest FFC of n.

An example is shown in Figure 2. It can be seen that a node may have multiple FFCs but only one MFFC. In this work, we are not interested in different FFCs of one node, but (M)FFCs of different nodes. Generally speaking, the MFFC of a node contains all logic dedicated to drive the node. For instance, in Figure 2, node 3 is not in the MFFC of 7 since 3 also drives 6, 8 (i.e., 6, 8 are "external fanouts" if 3 is in the cone). (M)FFCs have two important properties:

Property 1. *(a) Two FFCs of two different nodes (roots) are disjoint if one of the roots is not transitive fanin/fanout of the other. (b) If two FFCs of two different roots overlap, then the overlapping part contains the predecessor root.*

Property 2. *The MFFCs of different nodes are either subsets of each other or disjoint from each other, i.e., they cannot partially overlap.*

To derive Property 1, suppose there exists two FFCs of node n_1, n_2 respectively and there is a node m in both FFCs. From Definition 1, any path from m to a PO must pass both n_1 and n_2, so one of them is transitive fanout of the other. This proved Property 1a. Without loss of generality, suppose n_1 is the predecessor, then the path can be expressed as $m \to n_1 \to n_2 \to o$. From the definition of logic cone, all the nodes along $m \to n_2$ (and thus n_1) are in the FFC of n_2. This proved Property 1b. Property 2 is a corollary when applying Property 1 to MFFCs.

Property 2 inspires our parallel framework whose big picture is as follows. Instead of processing a cone for each node, we only process a set of disjoint MFFCs which forms a *partitioning* of the AIG. The reason is that the objective of refactoring is to restructure relatively large subgraphs (more local restructuring can be done by rewriting), and thus smaller MFFCs that are completely covered by others can be omitted. In this way, the replacement step can be performed for all cones in parallel *without data race*. As each node belongs to only one cone, there will be no conflict during deletion. The concurrent creation of nodes considering logic sharing can be handled by our parallel hash-table, as specified in Section III-E.

B. Algorithm Details of GPU Refactoring

a) Collapsing stage: Our GPU refactoring starts with a collapsing stage to identify a partitioning of the AIG into a set of disjoint FFCs, in a *level-wise parallel* fashion from POs to PIs. Level-wise parallel means that the cones at the same level[1] will form a batch and be processed concurrently, followed by another batch of cones at the next level, and so on. From the viewpoint of the collapsed network, the cones at level $i + 1$ are the fanins of the cones at level i. Figure 1a shows an example AIG containing two levels of cones.

In this stage, a frontier array is maintained that stores the roots of the FFCs to be identified at the next level (batch), initialized as the POs. One GPU thread is assigned for each node in the frontier to perform intra-cone traversal, as well as to obtain the cut associated with the identified cone. The traversal of an FFC is essentially a best-first search from the root towards PIs, that greedily expands a node which increases the current cut size as few as possible, and stops at nodes with external fanouts. After all traversals are finished, the cut node lists from each thread are gathered into a new global frontier array to be used in the next iteration, with duplicate nodes and PIs filtered out. This stage finishes when the frontier array is empty. Note that FFCs instead of MFFCs are identified in the intra-cone traversal, and we will elaborate on this issue in Section III-C.

b) Resynthesis and replacement stage: The local functions of all the identified cones are resynthesized through truthtable computation, Sum-of-Product generation and algebraic factoring, where one GPU thread is assigned for one local function. The gain of each new

[1]The level here is not the level of an AIG defined in Section II-A.

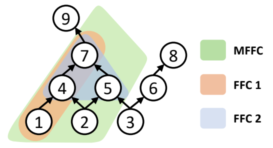

Fig. 2: An example for the MFFC and two FFCs of node 7.

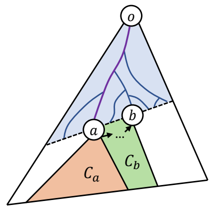
Fig. 3: Illustration for the disjointness of cones.

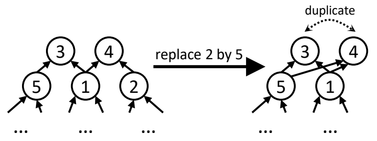

Fig. 4: Illustration for the occurrence of duplicate nodes.

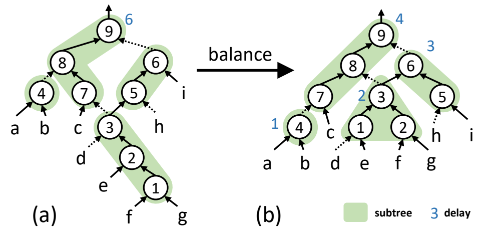

Fig. 5: An example for balancing.

cone is computed, and cones with negative gain are filtered out from replacement (Figure 1b). In the replacement stage, a parallel hash-table (see Section III-E) is initialized containing nodes in the cones not to be replaced, and all the cut nodes of the cones to be replaced (they are also old roots of some other cones; Figure 1c). The insertion of the new cones is performed iteratively through the sharing-aware node creation to be introduced in Section III-E, where one node for each cone is inserted concurrently in one iteration (Figure 1d-1e). Finally, the old roots are replaced by the new roots (Figure 1f).

C. Disjointness of the Identified Cones

The reason why FFCs, instead of MFFCs (as described in Section III-A), are identified in the collapsing stage (Section III-B) is because we need to early-stop a cone traversal when the associated cut size reaches the user-given maximum cut size. If this limit has never been reached in a traversal, the MFFC will be obtained. However, this does not harm the disjointness of the cones, as shown below.

Theorem 1. *The cones identified by the collapsing stage of GPU refactoring are all disjoint.*

Proof. Assume that two FFCs C_a and C_b rooted at node a and b constructed in the collapsing stage of GPU refactoring do overlap. According to Property 1, node a and b must be transitive fanin or fanout of each other and the overlapping part must contain the predecessor node. W.l.o.g., assume that b is the transitive fanout of a, i.e., a is the predecessor node and thus $a \in C_b$. As both C_a and C_b are constructed, both a and b have appeared in some frontier arrays. There are two cases. For case one, they appear in the frontier array at the same level, or a appears at an earlier level than b. In this case, C_a will be constructed no later than C_b, and thus a is reached via a path that does not pass b, as shown in Figure 3. Hence the construction of C_b will stop at (not including) node a since a must also fanouts to a node in that path. For case two, a appears at a later level than b. In this case, C_b will be constructed first. When we construct C_b, node a will not be included as this will contradict with the fact that a appears in a frontier array later on. Both cases contradicts with Property 1 that $a \in C_b$ and we can thus conclude that C_a and C_b do not overlap. □

D. Discussion on Gain Computation

Under the settings of parallel replacement, the original definition for the gain of a resynthesized cone (number of nodes decremented by replacing the cone individually) is no longer meaningful, since it cannot be uniquely determined which new cone a shared node should be assigned to when counting the number of added nodes. To address this issue, we re-define the gain of a new cone to be the difference between the number of deleted nodes and the nodes in the new cone, i.e., the logic sharing among new cones is omitted. Since the old cones are disjoint (Section III-C), the number of deleted nodes is accurately counted. Hence, our new gain is in fact a *lower bound* of the old gain, and thus there will be no area increase by replacing cones with non-negative new gains. In our implementation of the gain computation, we still consider the logic sharing between new cones and those nodes initialized in the hash-table, which makes our evaluation process semi-sharing-aware.

E. GPU-parallel Hash-table

In order to ensure node uniqueness when creating new AND nodes, we develop an efficient and versatile GPU-parallel hash-table that supports batched insertion and query of key-value pairs. Compared with the primitive hashing implementation in [9], our hash-table uses linear probing instead of chaining for handling conflicts and thus will benefit from memory locality more. It also supports dumping all the key-value pairs concurrently to a consecutively stored array, which is useful for further processing or result output. When creating a new AND node, a shareable node can be discovered by insertion followed by querying the same inserted key. If the retrieved value (id) is not the inserted one, there exists the same node in the hash-table whose id is given by the retrieved value and no new node will be inserted.

F. De-duplication and Dangling Node Removal

In practice, we found that the final AIG produced by parallel refactoring and GPU rewriting [9] may still contain duplicate and dangling nodes in some scenarios. If the new root of a resynthesized cone already exists in the AIG, duplication may occur among the fanouts of the old and new roots after replacement. Figure 4 shows an example in which 3 and 4 become a duplicate pair after replacing an old cone rooted at 2 by the cone rooted at 5. Dangling nodes occur when a local function does not depend on some of its input nodes. Although these are not common, we integrate an extra de-duplication and dangling node cleanup pass into our parallel refactoring as well as GPU rewriting as a final post-processing step. One GPU thread is assigned for each node with zero fanout to remove its MFFC. For de-duplicatation, AIG nodes are inserted and updated using our hash-table, in level-wise parallel from PIs to POs. The reason to process level-wise is that once some nodes are de-duplicated, there may occur new duplicate nodes among their fanouts, as illustrated in Figure 4 where de-duplicating 2, 5 creates new duplicated nodes 3, 4.

IV. GPU-parallel Balancing

A. ABC Balancing

Balancing reduces the delay of an AIG by a recursive subroutine. The subroutine starts with an entry node, and a cluster of nodes rooted at the entry node is identified that (1) forms a subtree and (2)

TABLE I: Normalized sequential part runtimes of three parallel algorithms. Reported times are averaged over the benchmarks in Table II.

Algorithm	GPU `rw` [9]	`rf` w/ seq. replace	`rf` (proposed)
Norm. seq. time	1.0	1.6	0.6

contains no internal complemented edges and multiple fanout nodes (e.g., cluster of 9, 8, 7 with root 9 in Figure 5a). From the functional viewpoint, this cluster is equivalent to an n-input AND node. The subroutine is then recursively invoked at all the inputs to this n-input AND node. Finally, the balanced inputs with optimized delays are iteratively combined by 2-input AND nodes in a delay-optimal order, i.e., nodes with smaller delays are combined first. For instance, in the subtree rooted at 9 in Figure 5b, c and 4 (with delay 0 and 1) are combined first, followed by 3 (delay 2), 6 (delay 3). In this paper, we use the term “subtree” and “n-input AND node” interchangeably.

B. Formulation of A Parallel-friendly Framework

ABC balancing has two main steps: identification (collapse) of n-input AND nodes, and local AIG (subtree) reconstruction with reordered inputs. Since the algorithm is recursive, these two steps may interleave with each other when different parts of the AIG are processed. For instance, in Figure 5a, the execution order is: collapse (the tree rooted at) 9, collapse 4, reconstruct 4, collapse 3, and so on. Such complexity, as well as the depth-first nature of the algorithm, impedes straightforward efficient parallelization.

Two key observations lead to our effective parallel balancing framework. First, since only the reconstruction step contributes to the structure of the resulting AIG, we can resolve the interleaving issue by separating the collapse and reconstruction for all subtrees into two separate stages. Second, balancing has an important property:

Property 3. *The reconstruction order of the collapsed subtrees does not affect the delay of the resulting balanced AIG, as long as the topological dependencies are satisfied, i.e., for any subtree, the subtrees rooted at its inputs need to be reconstructed first.*

To see this, we regard the AIG as a Boolean network in which each node is a collapsed n-input AND gate. The reconstruction step only manipulates local functions, so the graph structure of the network does not change after balancing, no matter what reconstruction order is used. Besides, it can be seen that the delay of each node is deterministic, since the order of combining its fanins is fixed and is the delay-optimal one (Section IV-A). The result is again irrelevant to when a fanin node is reconstructed.

Property 3 is important because it allows us to re-arrange the order of reconstruction of the subtrees in a parallel-friendly manner. Specifically, we adopt the *breadth-first* order, which enables the subtrees to be reconstructed in a level-wise parallel fashion. The collapse operation is also performed in level-wise parallel and is conceptually similar to the collapsing stage in GPU refactoring (Section III-B).

C. Algorithm Details of GPU Balancing

GPU balancing starts with the collapse operation which is essentially the same as the collapse in GPU refactoring, except that the identified cones are subtrees whose local functions are n-input AND gates, and there is no early-stopping during cone traversal. An empty hash-table is then initialized and the collapsed subtrees are reconstructed in level-wise parallel from PIs to POs. As introduced in Section IV-A, the reconstruction of a subtree is essentially combining its (already reconstructed) inputs in a delay-optimal order

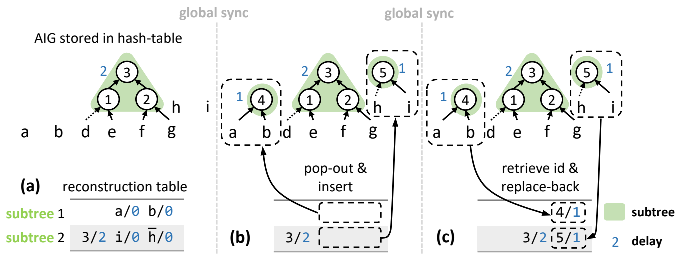

Fig. 6: One insertion pass of GPU balancing (cf. Figure 5b).

using 2-input AND nodes, which is conceptually similar to Huffman tree construction. In our algorithm, we use a reconstruction table to manage the intermediate nodes that remain to be combined for each subtree in the current batch (level), which is initialized with the fanin nodes of the subtrees (Figure 6a). During reconstruction, the insertion of new nodes into the hash-table (i.e., iteratively combining two intermediate nodes) is performed in parallel synchronously by a subroutine, called an *insertion pass*, that concurrently creates one new node for each subtree. Figure 6 illustrates an insertion pass for a batch of two subtrees rooted at 4 and 6 in the AIG shown in Figure 5. For each subtree, two nodes with minimum delays are retrieved, and the new node combining the two is created and returned to the reconstruction table for subsequent combination with other nodes. The reconstruction of a subtree batch is done by repeatedly applying insertion passes until there remains no intermediate nodes to be combined in the batch. Note that we cannot reconstruct all subtrees concurrently as in GPU refactoring, due to the topological constraint stated in Property 3.

V. Experimental Results

A. Set-up

We implement our proposed GPU refactoring and balancing in CUDA C/C++, and the source code of GPU rewriting is obtained from the author of [9]. The three algorithms are integrated into a single program. Experiments are performed on an Ubuntu 18.04 server with Intel Xeon Silver 4114 CPU and NVIDIA GeForce RTX 3090 GPU that has 24 GB dedicated DRAM.

We mainly adopt the EPFL Combinational Benchmark Suite [13] for evaluating our algorithms, and follow the approach in [8], [9] to select and enlarge the benchmarks since most of the cases in the benchmark suite are very small and thus the runtime will not benefit from parallelization. Since the selected benchmarks are mostly arithmetic circuits, we add two control circuits (also enlarged) from the OpenCores designs in the IWLS 2005 Benchmarks [14]. Three MtM benchmarks from the EPFL benchmark suite are also included in our evaluation, which are random Boolean functions instead of real circuits. The benchmark statistics are shown in Table II.

To our knowledge, there is no other parallel implementation of balancing and refactoring, so we only compare our algorithms with the sequential implementations in ABC. We use the ABC commands `balance`, `drf`, `drw` for balancing, refactoring and rewriting. The maximum cut size in refactoring is set as 11 for *log2* (due to insufficient thread-local memory) and 12 otherwise. The time reported only consists of the running time of optimization algorithms, which excludes the time of e.g., file I/O, initial structural hashing (in ABC). All the generated AIGs passed equivalence checking.

B. Results

a) Single optimization: We first apply our two algorithms and the ABC counterparts individually on the benchmarks, and the results

TABLE II: Result of single optimization algorithms by GPU (ours) vs. ABC.

Benchmarks	Statistics	ABC `balance`		GPU `b`		ABC `drf`		GPU `rf` (×2)	
	#Nodes / Levels	#Nodes / Levels	Time	#Nodes / Levels	Time	#Nodes / Levels	Time	#Nodes / Levels	Time
twentythree	23339737 / 176	17518692 / 104	59.4	17513406 / 104	1.8	18126777 / 117	1069.4	18397973 / 103	10.7
twenty	20732893 / 162	15646118 / 94	54.6	15641552 / 94	1.6	16138461 / 110	948.5	16421342 / 95	9.2
sixteen	16216836 / 140	12250387 / 99	35.0	12246434 / 99	1.4	12617619 / 104	737.7	12911473 / 101	7.6
div_10xd	58620928 / 4372	58612736 / 4372	177.4	58598400 / 4372	20.1	57035776 / 4374	449.5	48779264 / 4422	20.5
hyp_8xd	54869760 / 24801	54869760 / 24801	185.0	54869760 / 24801	86.6	54539008 / 24801	362.3	54539008 / 24790	32.7
mem_ctrl_10xd	47960064 / 114	47959040 / 114	161.3	47959040 / 114	9.2	47592448 / 114	907.8	47444992 / 109	16.9
log2_10xd	32829440 / 444	32706560 / 410	93.3	32697344 / 410	5.0	31441920 / 425	344.8	31552512 / 396	7.9
multiplier_10xd	27711488 / 274	27599872 / 266	82.9	27599872 / 266	5.9	26664960 / 272	229.6	26565632 / 265	5.5
sqrt_10xd	25208832 / 5058	25204736 / 5058	67.6	25204736 / 5058	9.3	24033280 / 5063	207.7	24862720 / 5365	12.4
square_10xd	18927616 / 250	18748416 / 250	62.9	18740224 / 250	3.6	18142208 / 250	144.8	18081792 / 250	3.7
voter_10xd	14088192 / 70	13907968 / 70	41.8	13748224 / 70	3.4	11744256 / 62	98.7	11480064 / 66	2.5
sin_10xd	5545984 / 225	5521408 / 186	16.9	5516288 / 186	0.9	5342208 / 224	53.8	5357568 / 213	1.5
ac97_ctrl_10xd	14610432 / 12	14597120 / 11	62.2	14597120 / 11	2.8	11825152 / 12	201.5	11144192 / 12	3.4
vga_lcd_5xd	4054752 / 24	4053696 / 19	16.7	4053696 / 19	1.1	3236384 / 24	63.2	2952480 / 26	1.2
Geomean Ratio vs. ABC		1.000 / 1.000	1.0	0.999 / 1.000	14.8× accel.	1.000 / 1.000	1.0	0.983 / 0.980	42.7× accel.

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

TABLE III: Result of optimization sequences by GPU (ours) vs. ABC.

Benchmarks	ABC `rf_resyn`		GPU `rf_resyn`		ABC `resyn2`		GPU `resyn2` (`rwz` ×2)	
	#Nodes / Levels	Time	#Nodes / Levels	Time	#Nodes / Levels	Time	#Nodes / Levels	Time
twentythree	17396577 / 104	1490.4	17348255 / 98	17.3	16931332 / 72	4174.5	16915942 / 68	55.2
twenty	15514368 / 94	1359.7	15480873 / 90	15.2	15095643 / 65	3633.5	15079948 / 65	49.1
sixteen	12147445 / 99	1115.4	12118520 / 99	13.2	11765351 / 68	2760.7	11757432 / 64	40.4
div_10xd	56788992 / 4404	2273.1	48652485 / 4373	104.7	41665780 / 4388	8028.8	41689305 / 4422	239.8
hyp_8xd	54539008 / 24785	2104.8	54539008 / 24787	345.2	54193719 / 24785	10771.1	54205696 / 24671	653.8
mem_ctrl_10xd	46615552 / 105	3448.9	47312818 / 108	54.7	43777052 / 92	6936.8	44821695 / 94	132.9
log2_10xd	31011840 / 366	1434.7	31371816 / 390	34.3	29946093 / 358	4879.4	29966717 / 358	91.5
multiplier_10xd	26471424 / 265	947.1	26469376 / 265	33.0	24957961 / 262	3509.3	24949760 / 262	77.5
sqrt_10xd	23618560 / 5182	3904.8	23014400 / 5174	54.8	18884491 / 6020	5639.8	18800640 / 5928	131.5
square_10xd	17935360 / 250	606.7	17888256 / 250	22.9	17091593 / 248	2343.3	17052614 / 249	58.0
voter_10xd	9951232 / 59	370.6	10874377 / 63	14.6	8845336 / 66	1432.4	8961831 / 60	33.2
sin_10xd	5285888 / 179	232.0	5319346 / 187	6.2	5156077 / 163	851.0	5158131 / 161	16.2
ac97_ctrl_10xd	10956800 / 11	699.6	11020147 / 9	13.5	10490213 / 10	1375.5	10660403 / 10	32.0
vga_lcd_5xd	2918528 / 18	189.5	2946898 / 20	5.2	2903540 / 24	439.6	2903968 / 23	10.7
Geomean Ratio vs. ABC	1.000 / 1.000	1.0	0.996 / 1.000	39.5× accel.	1.000 / 1.000	1.0	1.003 / 0.982	45.9× accel.

are shown in Table II. For balancing, our GPU algorithm (denoted by `b`) achieves $14.8\times$ acceleration on average over ABC. Since GPU balancing works in level-wise parallel, it obtains higher acceleration on shallower AIGs than on deeper AIGs. This is understandable, because for two AIGs with a similar number of nodes, the one with larger level has smaller degree of parallel in each level. On all benchmarks, the levels of the GPU balancing results are the same as those of ABC, which is theoretically guaranteed by Property 3.

For refactoring, since the sequential algorithm updates local subgraphs on-the-fly in topological order, subsequent updates can benefit from previously resynthesized cones. In parallel refactoring (as well as GPU rewriting), there is no such benefit since all cones are resynthesized simultaneously, which affects the result quality but it can be caught up by repetition. Hence, we perform two passes of GPU refactoring (denoted by `rf`). Under this setting, GPU refactoring still obtains a significant acceleration of $42.7\times$, as well as 1.7% smaller area and 2.0% smaller delay over ABC results. Parallel replacement contributes a lot to such efficiency. As shown in Table I, the runtime of the sequential part in GPU refactoring is very small, which only consists of some post-processing procedures, and there is a high level of parallelism in our replacement step.

Since some of the cones replaced in GPU refactoring might have zero gain, for fair comparison, we also summarize our results (GPU `rf` $\times 2$) versus ABC refactoring with zero gain replacement enabled (`drf -z`): GPU refactoring achieves $61.0\times$ acceleration with 3.3% smaller area and 2.5% larger delay.

b) Optimization sequence: We then inspect the proposed algorithms under the settings of optimization sequences. This is more common in practice than only applying a particular algorithm once or repeatedly, as introduced in Section I. Since most of the optimization sequences contain rewrite (`rw`), we create a new sequence named `rf_resyn` by replacing `rw` in `resyn` with `rf`, in order to exclusively reflect the runtime efficiency and result quality of our two proposed algorithms. `rf_resyn` is specified as: `b; rf; rfz; b; rfz; b`, where the suffix `z` stands for accepting zero gain replacements. As stated previously, GPU refactoring by default accepts zero gain replacements since the computed gain is a lower bound. So, `rfz` is only meaningful for ABC, while `rf` and `rfz` make no difference for GPU refactoring. The results in Table III confirm that the performance of our two algorithms is consistent with that in the single algorithm case, with an acceleration ratio of $39.5\times$. In particular, GPU refactoring is only applied once instead of twice for each `rf(z)` command, and this is already sufficient for GPU `rf_resyn` to improve over the quality of ABC.

We further perform experiments on the well-known `resyn2` sequence: `b; rw; rf; b; rw; rwz; b; rfz; rwz; b`, by

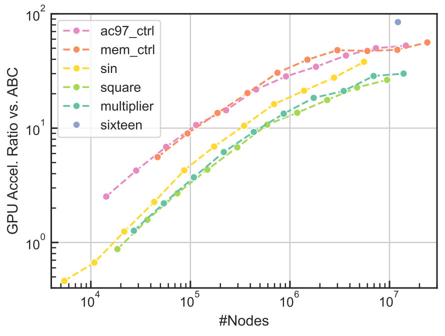

Fig. 7: Acceleration ratios of GPU `rf_resyn` over ABC on AIGs with different sizes.

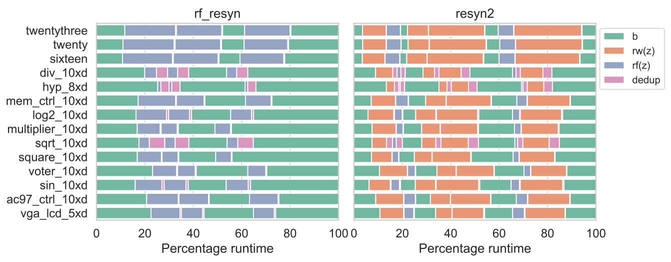

Fig. 8: Runtime breakdown of GPU `rf_resyn` and `resyn2`.

integrating our algorithms with GPU rewriting [9]. Note that in GPU `resyn2` we apply two passes of rewriting with zero gain replacement[2] for each `rwz`, but one pass of the corresponding algorithm for all the other commands. As shown in Table III, `resyn2` is accelerated $45.9\times$ by GPU with marginally larger area but 1.8% better delay.[3] To conclude, our GPU algorithms successfully speed up AIG optimization flows by $40\times$ to $46\times$ without quality degradation.

In order to obtain insights into the runtime efficiency of our algorithms with different problem sizes, we enlarge benchmarks to different scales and check the acceleration ratios of GPU `rf_resyn` over ABC. Figure 7 shows the result. The acceleration ratio increases consistently with increasing problem size. When the number of nodes is less than 30k, GPU `rf_resyn` may be slower than ABC, due to GPU operation overheads. However, since modern chips can contain billions of transistors, it is very promising for our GPU logic optimization flow to achieve high acceleration on these large designs.

Figure 8 shows the runtime breakdown of each command in GPU sequences. The runtime of de-duplication and dangling node removal (denoted by `dedup`) is separated from `rw` and `rf`. It can be seen that `b` occupies a large proportion of total runtime (especially in `rf_resyn`) though sequential balancing is much more efficient than sequential rewriting and refactoring, indicating the significant speedup by GPU `rw` and `rf`. In particular, for benchmarks with large delays, the runtimes of `b` and `dedup` become significant, due to their level-wise parallel nature.

VI. Conclusion

In this paper, we presented GPU-parallel refactoring and AND-balancing for AIG optimization. We proposed a novel refactoring framework in which the replacements of local subgraphs are performed in parallel without data race. Specifically, a partition of the AIG consisting of only fanout-free cones is obtained and all cones are resynthesized and replaced simultaneously. For GPU AND-balancing, the AIG is processed level-by-level, where subtree structures of the AIG within the same level are identified and reconstructed concurrently. Furthermore, we enabled the `resyn2` sequence to be fully GPU-accelerated, by integrating our two algorithms with GPU rewriting. Experiments showed that our GPU refactoring, AND-balancing, and `resyn2` implementation on CUDA-enabled GPU achieved acceleration ratios of $42.7\times$, $14.8\times$ and $45.9\times$ over ABC with comparable or better quality of results. Future work includes parallelizing the resynthesis process for each cone or introducing new resynthesis methods in refactoring, and parallelizing more logic optimization algorithms such as resubstitution.

[2]Zero gain replacement is not considered in [9], so we slightly modified their code on some conditions without changing the algorithmic framework.

[3]Due to the system-related non-determinism in CUDA thread execution order, there is a very slight variation in the area of the results, measured to be less than 0.001% on average (by repeating GPU `resyn2` five times). No variation in the delay of the results is observed. This phenomenon is also reported in [8] for parallel rewriting.

Acknowledgment

This research was partially supported by a grant from the AI Chip Center for Emerging Smart Systems Ltd.

References

[1] A. Kuehlmann, V. Paruthi, F. Krohm, and M. Ganai, “Robust boolean reasoning for equivalence checking and functional property verification,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 21, no. 12, pp. 1377–1394, 2002.

[2] A. Mishchenko, R. Brayton, S. Jang, and V. Kravets, “Delay optimization using SOP balancing,” in *2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2011, pp. 375–382.

[3] J. Cortadella, “Timing-driven logic bi-decomposition,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 22, no. 6, pp. 675–685, 2003.

[4] A. Mishchenko, S. Chatterjee, and R. Brayton, “DAG-aware AIG rewriting: a fresh look at combinational logic synthesis,” in *2006 43rd ACM/IEEE Design Automation Conference (DAC)*, 2006, pp. 532–535.

[5] A. Mishchenko and R. K. Brayton, “Scalable logic synthesis using a simple circuit structure,” in *2006 15th International Workshop on Logic and Synthesis (IWLS)*, 2006.

[6] R. K. Brayton and A. Mishchenko, “ABC: An academic industrial-strength verification tool,” in *2010 22nd International Conference on Computer Aided Verification (CAV)*, 2010.

[7] S. Chatterjee, A. Mishchenko, R. Brayton, X. Wang, and T. Kam, “Reducing structural bias in technology mapping,” in *2005 IEEE/ACM International Conference on Computer-Aided Design*, 2005, pp. 519–526.

[8] V. Possani, Y.-S. Lu, A. Mishchenko, K. Pingali, R. Ribas, and A. Reis, “Unlocking fine-grain parallelism for AIG rewriting,” in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2018, pp. 1–8.

[9] S. Lin, J. Liu, T. Liu, M. D. Wong, and E. F. Young, “NovelRewrite: Node-level parallel AIG rewriting,” in *2022 59th ACM/IEEE Design Automation Conference (DAC)*, 2022.

[10] Y. Lin, S. Dhar, W. Li, H. Ren, B. Khailany, and D. Z. Pan, “DREAMPlace: Deep learning toolkit-enabled GPU acceleration for modern VLSI placement,” in *2019 56th ACM/IEEE Design Automation Conference (DAC)*, 2019, pp. 1–6.

[11] S. Lin, J. Liu, and M. D. Wong, “GAMER: GPU accelerated maze routing,” in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2021, pp. 1–8.

[12] R. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang, “MIS: A multiple-level logic optimization system,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 6, no. 6, pp. 1062–1081, 1987.

[13] L. Amarú, P.-E. Gaillardon, and G. De Micheli, “The EPFL combinational benchmark suite,” in *Proceedings of the 24th International Workshop on Logic and Synthesis (IWLS)*, 2015.

[14] C. Albrecht, “IWLS 2005 Benchmarks,” https://iwls.org/iwls2005/benchmarks.html, 2005.