GPU-Accelerated Efficient Transduction for Logic Optimization

Zhuofan Lin
Thrust of Microelectronics
HKUST Guangzhou
zlin037@connect.hkust-gz.edu.cn

Shiju Lin
Thrust of Microelectronics
HKUST Guangzhou
shijulin@hkust-gz.edu.cn

***Abstract*—Transduction is a powerful method for high-effort logic optimization. Unlike many local heuristics that focus on area-decreasing steps, transduction incorporates area-increasing transformations to restructure circuits, thereby uncovering unique opportunities for subsequent area reductions. Despite its potential in area optimization, transduction is computationally expensive, primarily due to the high runtime cost of computing don't-cares. To reduce its runtime and make it more practical, we present a GPU-accelerated fast transduction algorithm. We first explore how to maximize the parallelism of transduction, followed by GPU-friendly kernel optimization techniques for reduced memory consumption and improved performance. Compared to the state-of-the-art transduction implementation in ABC, our method achieves an average speedup of 130× while delivering superior and-inverter graph (AIG) results on the large benchmarks from the IWLS2022 Programming Contest. The source code of this work is available at https://github.com/Lin-HKUST-Guangzhou/gpu-transduction.**

I. Introduction

In the design flow of integrated circuits, logic synthesis is a pivotal process that transforms register-transfer level (RTL) descriptions into optimized gate-level netlists. A key component of this process is *logic optimization*, which refines the design based on a generic intermediate representation, such as And-Inverter Graph (AIG). One primary goal during logic optimization is to minimize the number of AIG nodes, thereby reducing the design area. This reduction is useful not only for area-constrained designs, but also for timing-critical designs, as a smaller area can facilitate further timing optimization, even if it results in an area increase.

To tackle increasingly large and complex designs, the focus of logic optimization has shifted towards fast, local-transformation-based algorithms, including *rewriting* [1]–[4], *resubstitution* [5]–[7], and *refactoring* [8]. While each algorithm individually may be suboptimal, interleaving these heuristics can lead to significant area reductions, as demonstrated by the `resyn2` process in ABC [9]. However, even such an iterative approach can still become trapped at local minima easily, because all these algorithms focus only on area-reducing transformations.

To overcome this limitation, the concept of *transduction*, originally introduced in the 1980s [10], has been revisited and

Corresponding author: *Shiju Lin*

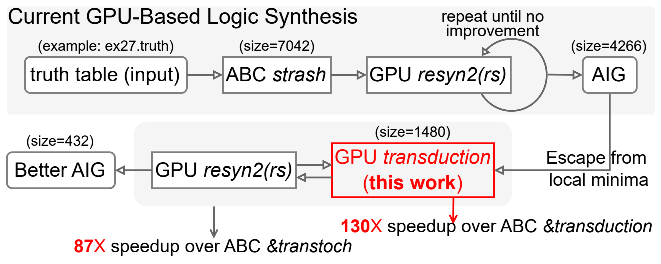

Fig. 1: Our GPU-accelerated transduction achieves 130× speedup over the state-of-the-art transduction [11], enhancing the capabilities of open-source GPU-based logic synthesis.

modernized (`&transduction` in ABC) [11]. This new version leverages observability don't cares (ODCs) to introduce additional wires without altering the functionality, a process known as *transformation*. This is followed by *reduction*, which removes constant edges and nodes based on ODCs. Since each AIG node can have a maximum of two fanins, the *transformation* phase, which involves adding wires, effectively increases the number of nodes. This unconventional AIG restructuring presents a unique optimization opportunity for the *reduction* phase. Although *transformation* initially increases the area, the subsequent *reduction* may lead to a substantial decrease, resulting in an enhanced overall solution.

Despite its potential for area optimization, transduction faces challenges due to its extremely high runtime, which limits its practicality and scalability. For instance, processing an 18-input AIG with 10,000 nodes can take over three days using current methods. Recent advancements in GPU computing for various EDA tasks [12]–[33] have motivated us to investigate GPU acceleration for transduction. However, developing an efficient, GPU-parallel transduction algorithm presents two main challenges.

Challenge 1: The current CPU transduction framework is unsuitable for GPU implementation. The state-of-the-art CPU transduction framework [11] utilizes both multi-input AIG (MIAIG) and Binary Decision Diagram (BDD), with BDDs used for functional manipulation involving ODC. Managing these dual logic representations on a GPU would require substantial precise GPU memory and more sequential, difficult-to-parallelize control-flow operations to synchronize

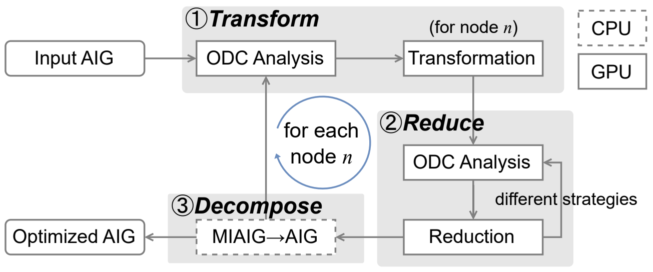

Fig. 2: Our transduction flow

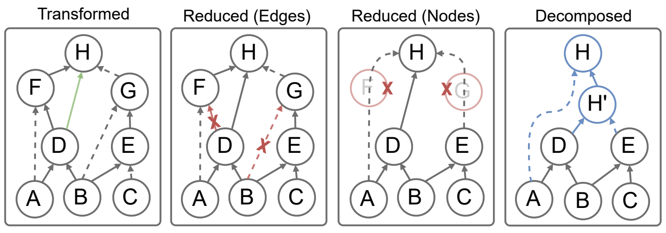

Fig. 3: Illustration of different stages in Figure 2

TABLE I: Explanation of Variables Used in This Paper

Variable	Explanation
`N`	number of primary inputs
`p`	an input pattern
`PI/PO`	set of all primary inputs/outputs
`fanins`(x)	set of all fanins of node x
`val`(n, p)	logic value of node n under p
`isODC`(n, p)	whether node n is ODC under p
`isEdgeODC`(n, fi, p)	whether edge(n, fi) is ODC under p
`isEdgeConst`(n, fi, v)	whether edge(n, fi) is constant v

between the AIG and BDD. This creates significant challenges in memory management and efficiency if the GPU-parallel approach directly adapts the existing CPU version.

Challenge 2: Transduction has unique computational patterns that have not been optimized for GPU acceleration. Current GPU-accelerated logic optimization algorithms, such as rewriting [22] and resubstitution [23], focus on local transformations, with computational demands roughly proportional to the number of AIG nodes. In contrast, transduction relies on ODCs and involves computations that scale exponentially with the number of primary inputs. This substantial computational workload, combined with unexplored computational patterns, requires novel parallelization methods that are fundamentally different from existing GPU-parallel approaches [22], [23], [34].

To tackle these challenges, we propose a GPU-based transduction method with the following key contributions:

- We introduce an MIAIG-based, parallel transduction framework, as shown in Figure 2. This framework modularizes the transduction process, enabling most modules to run on the GPU and allowing for module reuse (e.g., *ODC Analysis*) to maximize the efficiency with minimized development efforts.
- We develop customized, fine-grained parallelization strategies for individual modules within the transduction flow to maximize parallelism. Additionally, we propose a depth-aware heuristic to minimize increase of AIG depth.
- We implement a bit-packing technique to reduce GPU memory consumption and propose two optimizations to address efficiency challenges associated with bit packing.

Experimental results demonstrate that our method achieves a 130× speedup over the state-of-the-art CPU-based implementation [11], while also improving size and depth. Furthermore, a new flow that integrates our method with existing GPU-based logic optimization algorithms is 87× faster than a high-effort, transduction-based flow in ABC. This work will be open-sourced.

II. Preliminaries

A. (Multi-Input) And-Inverter Graphs

An *and-inverter graph* (AIG) is a type of directed acyclic graph (DAG) that is used to represent Boolean logic networks. In an AIG, the nodes are categorized into two-input nodes, performing logical conjunctions (AND operations), and zero-input nodes, known as *primary inputs* (PIs). The edges of the graph may include inverters to denote logical negation. Some nodes are connected to *primary outputs* (POs), which correspond to the outputs of the logic network. The *size* of an AIG is the number of nodes. The *level* of a node is the maximum number of nodes in a path from a PI to the node. The *depth* of an AIG is the largest level of its POs.

A *multi-input and-inverter graph* (MIAIG) [11], on the other hand, extends the traditional AIG by allowing each AND node to accept multiple inputs. This enhancement provides greater flexibility, which is essential for the transduction method.

B. Observability Don't-Cares

Under a specific combination of PIs' values (referred to as an *input pattern* p), the value of a node n may be *unobservable* by any POs, i.e., changes in the logic value of node n do not change the value of any POs. The input pattern p is called an *observability don't care* (ODC) for node n. ODCs allow us to modify the Boolean functions of internal nodes while preserving the AIG's functionality, providing great flexibility and potential for optimization. We compute compatible sets of ODCs instead of maximum sets, which are used in previous work [11] most of the time.

III. Massively Parallel Transduction

Table I explains the notations used in this paper.

A. Overview

Figure 2 illustrates our iterative transduction process. Each iteration begins by selecting a node for the *Transform* operation, followed by executing *Reduce* and *Decompose* on the entire AIG. This iterative process ensures that each node is selected exactly once, following a topological order. Below, we detail an iteration using the example provided in Figure 3.

Algorithm 1: Parallel ODC Analysis

```
Data: MIAIG, input patterns $P = \{0,1\}^{\mathrm{N}}$
Result: isODC, isEdgeODC
1  Ⓐ Levelization
   // Ⓑ Exhaustive Simulation (lines 2–4)
2  for $l \leftarrow 0$ to $maxLevel$ do
3      for each node $n$ of level $l$ and each $p \in P$ in parallel do
4          calculate val$(n, p)$
   // Ⓒ ODC Calculation (lines 5–17)
5  Initialize all isODC and isEdgeODC to true
6  Initialize isODC$(po, *)$ to false for all $po \in$ PO
7  for $l \leftarrow maxLevel$ to 0 do
8      for each node $n$ of level $l$ and each $p \in P$ in parallel do
9          if isODC$(n, p)$ = false then
10             select $care \in \{x \in$fanins$(n) |$val$(x, p) = 0\}$
11             if $care \neq$none then
12                 isEdgesODC$(n, care, p) \leftarrow$ false
13             else
14                 isEdgesODC$(n, *, p) \leftarrow$ false
15             for each $fi \in$ fanins$(n)$ do
16                 if isEdgeODC$(n, fi, p)$ =false then
17                     isODC$(fi, p)$ ←false
```

① ***Transform***: This initial step identifies potential additional connections that can be established without altering the Boolean functions of the POs. In the example from Figure 3, node H is selected for transformation during this iteration. Using results from *ODC Analysis*, we determine that connecting node D to node H (green arrow) is a valid transformation that preserves the functionality of the AIG.

② ***Reduce***: The subsequent step performs ODC-based *reduction* to eliminate redundant edges and nodes. In Figure 3, we identify two redundant edges, between node D and node F, and node B and node G, highlighted in red. Once these edges are removed, nodes F and G are left with only one fan-in each, making them redundant. Consequently, these nodes are also removed from the graph.

③ ***Decompose***: Finally, we decompose the MIAIG into a standard AIG by breaking multi-fanin AND nodes into cascaded two-fanin AND nodes. This decomposition facilitates precise AIG size calculation, enabling us to discard changes from less promising iterations. For instance, in Figure 3, the 3-fanin node H is decomposed into two 2-fanin nodes, H and H', resulting in an overall reduction of one node. If H' already exists in the original AIG, the node reduction increases to two.

B. ODC Analysis

The *ODC analysis* module computes simulated logic values and Observability Don't Care (ODC) results, serving as a crucial prerequisite for both *transformation* and *reduction* processes. As outlined in Algorithm 1, ODC analysis comprises three key steps: Ⓐ *levelization* (line 1), Ⓑ *exhaustive simulation* (lines 2–4), and Ⓒ *ODC calculation* (lines 5–17).

Levelization involves sorting nodes into levels using the recursive relationship defined as follows:

$$level(x) = \begin{cases} 0, & \text{if } x \in \texttt{PI} \\ 1 + \max_{y \in \text{fanins}(x)} level(y), & \text{otherwise} \end{cases} \quad (1)$$

Nodes at the same level have no dependencies among each other, allowing for concurrent processing. This level-based parallelism is used in both subsequent steps: *exhaustive simulation* and *ODC calculation*.

Exhaustive Simulation (lines 2–4) computes the logic values of each node for all possible 2^{N} input patterns. The logic value of a PI is determined by the input pattern, while the logic value of an AND node is determined by the logical conjunction of its fanins' logic values. As shown in line 2, we iterate over levels from 0 to the maximum level, calculating the logic values for each node and pattern in parallel (line 3). Figure 4 illustrates this parallelization. For instance, if there are 100 nodes at a level, we allocate $100 \times 2^{\mathrm{N}}$ GPU threads for simultaneous simulation.

ODC Calculation (lines 5–17) identifies the ODCs of each node and edge in reverse topological order. Initially, every node and edge is considered ODC (line 5), except for POs that can never be ODC (line 6). For each level l in reverse order (line 7), we process each node at level l with each input pattern in parallel. Processing a node n involves analyzing the ODC of its fanin nodes and edges, as follows:

- **Case 1:** If node n is ODC under pattern p, all its fanin nodes and edges are also ODC. Since `isEdgeODC` and `isODC` are initialized to true, no modification is needed.
- **Case 2:** If n is not ODC under pattern p (lines 9–17), let S be the set of n's fanins with a value of 0 for pattern p.
  - **Case 2.1:** If S is not empty, select a *care* fanin from S (line 10), making the other fanins ODC (line 12) since the logic value of AND node n is always 0, determined by the 0-value *care* fanin and independent of other fanin values.
  - **Case 2.2:** If S is empty, no fanin edges of n are ODC. Set all `isEdgeODC` for node n and pattern p to `false` (line 14).
  - **Final Step:** For non-ODC fanin edges of n, set their corresponding nodes to be non-ODC (lines 15–17).

There are multiple methods for selecting the *care* fanin in Case 2.1. We employ two strategies from previous work [11]: selecting the fanin with the smallest index or randomly.

C. Transformation

The *transformation* step involves adding new edges between nodes. Although this operation does not change the size of the MIAIG itself, it increases the size of the AIG derived from the MIAIG, enabling unconventional structural changes that may uncover new optimization opportunities. To add a new edge from node a to node b (i.e., making a a fanin of b), the following two conditions must be satisfied:

1) Node a must not be a transitive fanout of node b, meaning there should be no paths from b to a. If this condition is not met, a loop would be created, resulting

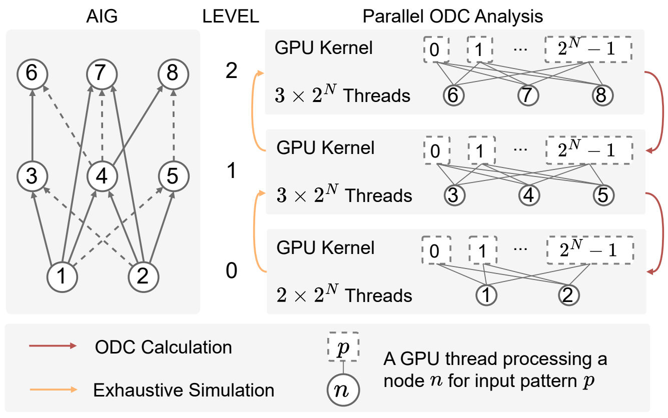

Fig. 4: Level- and input-pattern-based parallelism

in incorrect logic. To efficiently verify this condition, we utilize a level-based parallelization scheme. For each level $l \in [0, maxLevel]$, we examine each node at this level in parallel to determine if it is a transitive fanout of node b by checking whether any of its fanins is a transitive fanout of b.

2) For every input pattern p, the condition $\texttt{val}(a, p) = 0$ and $\texttt{val}(b, p) = 1$ must not occur simultaneously. To expedite this check, we allocate 2^{N} GPU threads, with each thread responsible for checking a different p.

Depth-Aware Heuristic. To prevent a significant increase in the depth of the final AIG when adding edges from higher-level nodes to lower-level nodes, we propose a depth-aware heuristic for the *transformation* process. For a node n, we consider only the half of the nodes in the MIAIG that are closest to n in topological order as candidates for fanins during transformation. This strategy prioritizes selecting fanins with comparable depths to minimize depth increases, while ensuring a sufficient number of candidates for transformation.

D. Reduction

The *reduction* step aims to identify and remove edges that can be considered constant. For instance, if an edge consistently holds a logic value of 1 across all input patterns, excluding those that are ODCs, it is deemed a constant 1 and can be removed. This requires examining all 2^{N} `isEdgeODC` results for each edge. When these 2^{N} results are checked *sequentially*, the process can terminate as soon as both logic values 0 and 1 are found, indicating that the edge cannot be constant. Since most edges are not constant and will trigger early termination, this mechanism effectively reduces redundant computations and enhances efficiency.

To leverage both parallelism and the early-termination mechanism concurrently, we have designed a parallel constant check kernel, as illustrated in Algorithm 2. For each edge, a block of 1024 threads is allocated, with each thread responsible for examining $2^{\mathrm{N}}/1024$ input patterns. This workload distribution offers the following benefits:

- **Parallelism:** A block of 1024 threads is allocated per edge, allowing for full utilization of GPU resources. As long as the AIG has more than 10 edges, all GPU cores are actively engaged (assuming a typical GPU has around 10k cores), maximizing resource utilization.
- **Early Termination:** Each thread sequentially checks $2^{\mathrm{N}}/1024$ input patterns. This workload is substantial enough to ensure that when early termination occurs, a significant portion of the uncompleted work can be discarded, further enhancing efficiency.

Algorithm 2: Parallel Constant Check

```
Data: MIAIG $G$, input patterns $P = \{0,1\}^{\mathrm{N}}$, isEdgeODC
Result: isEdgeConst
// This thread block checks edge $(n, fi)$
1 Initialize all isEdgeConst to true
2 for subset $P' \subset P$ in parallel do
3     for pattern $p \in P'$ do
4         if isEdgeODC$(n, i, p)$ = false then
5             $v \leftarrow$ val$(n, fi, p) \oplus 1$
6             isEdgeConst$(n, fi, v) \leftarrow$ false
7     if both isEdgeConst$(n, fi, 0/1)$ = false then
8         Terminate
```

IV. Bit Packing and Optimization

The previous section introduced our maximally parallel transduction framework. However, maximizing theoretical parallelism in transduction does not always translate into optimal performance in practice, primarily due to the unique characteristics of GPU architecture. In this section, we enhance the transduction framework to improve memory and runtime efficiency using a technique called bit packing.

A. Bit Packing

Bit packing involves compressing multiple Boolean variables into a single multi-bit integer. In this work, we represent 32 Boolean values using a 32-bit integer. For example, we encode the Boolean simulation results, `val`, of a node n for input patterns $p = 0, \ldots, 31$ using a 32-bit integer v. In this representation, $\texttt{val}(n, p)$ corresponds to the p-th bit of v. This compact representation provides several advantages:

- **Reduced Memory Consumption.** Although a Boolean type conceptually requires only one bit, a `bool` type in `CUDA/C++` occupies 1 byte (8 bits). Consequently, bit packing can significantly reduce memory usage, which is crucial given the limited GPU memory compared to CPU memory. Additionally, bit packing minimizes unnecessary data movement (since 7 out of 8 bits in a `bool` variable are redundant), thereby enhancing the GPU performance.
- **Efficient Bitwise Operations.** 32-bit integers are native data types in `CUDA` and support efficient bitwise operations. For instance, with 32 input patterns, simulating the logic value of an AIG node can be efficiently executed with a single bitwise AND operation instead of 32 separate Boolean AND operations.

While bit packing offers several advantages, it also presents challenges that must be addressed for effective utilization.

B. Challenge #1: Serialized Modifications for Packed Bits

When 32 Boolean variables are stored in a single 32-bit integer, updating any of these Boolean values requires a write operation to the memory address of the integer in `CUDA`. Simultaneous write operations can lead to race conditions, necessitating serialized operations (using `atomic` functions) to maintain correctness. Such serialization significantly reduces parallelism and negatively affects performance. For example, in an AIG with 18 PIs and 4266 nodes, bit packing slows down *ODC calculation* by over $8\times$.

Optimization #1. We implement two changes to address this challenge. First, we use a single thread to compute 32 Boolean results represented by a single 32-bit integer. Since serialized modifications are unavoidable, consolidating these operations into a single thread better utilizes GPU resources. Second, we allocate thread-local copies for `isODC` and `isEdgeODC`, compute their results locally, and update the global results once at the end of the thread. This approach reduces the need for slow global memory access by leveraging the faster thread-level local memory.

C. Challenge #2: Leveraging Bitwise Operations in Complex Computations with Branching

To fully capitalize on bit packing, computations should be expressed using bitwise operations whenever possible. However, many steps in transduction involve not only complex computations but also branching conditions, complicating the effective use of bitwise operations. For example, in lines 4–6 of Algorithm 2, an `if` condition in line 4 prevents us from utilizing bitwise operations for the computation in line 6. Moreover, the computed values v in line 5 are used as an index for `isEdgeConst` in line 6.

Optimization #2. To address the `if` condition, we use bitwise AND operations to filter out scenarios that do not meet our criteria (`false` for the `if` condition). To manage computed values used as indices, we separately handle cases when the value is 0 or 1.

Our approach to the above example is shown in Algorithm 3. First, we perform a bitwise XOR operation between `isEdgeODC`(n, i, p) and $2^N - 1$ (whose binary representation consists of all 1s) to extract bits of interest stored in `care`. Next, we perform a bitwise AND between `care` and `val` to find all the bits with a value of 1 and update `isEdgeConst` accordingly. These bits correspond to all input patterns p in the original Algorithm 2 where the `if` condition in line 4 is satisfied and v in line 5 equals 0. Similarly, for the other case, we find all relevant bits with a value of 0 using a bitwise AND and update `isEdgeConst` accordingly.

Algorithm 3: Bitiwse Operations for Constant Check

```
1 care ← isEdgeODC($n, i, p$) ⊕ ($2^N - 1$)
2 if care & val($n, fi, p$) then
3     isEdgeConst($n, fi, 0$) ← false
4 if care & (val($n, fi, p$) ⊕ ($2^N - 1$)) then
5     isEdgeConst($n, fi, 1$) ← false
```

V. Experimental Results

A. Setup

The proposed GPU-accelerated transduction method is implemented in CUDA and was evaluated on an Ubuntu 22.04 server equipped with AMD EPYC 7542 CPUs and an NVIDIA GeForce RTX 4090 GPU with 48G DRAM.

For evaluation, we used the truth-table benchmarks from the IWLS 2022 Programming Contest[1], which were also employed by the previous transduction research [11]. These benchmarks were converted into AIGs using ABC `strash` [9]. We selected AIGs with at least 16 PIs and a minimum of 3000 nodes, as these sizes present a significant challenge to the efficiency of transduction methods. We iteratively applied all existing GPU-based logic optimization algorithms [22], [23], [34] to these large AIGs until convergence. Specifically, we interleaved the `resyn2` and `resyn2rs` optimization sequences from the GPU-based logic synthesis tool CULS[2] until no further improvements could be made. This process generates local minima results where state-of-the-art GPU-based logic optimization tool reaches its limit, highlighting a potential application of our GPU-accelerated transduction method as part of a more comprehensive GPU-based synthesis tool. Details of the resulting AIGs are provided in the "Statistics" column of Table II.

B. Transduction Results

First, we conducted experiments to compare the state-of-the-art transduction [11] (`&transduction` in ABC) with our GPU-accelerated transduction method. The results, presented in the "Single Transduction" column of Table II, demonstrate that our parallel approach significantly outperforms the CPU implementation. Specifically, our method produces better AIGs in terms of both size and depth, achieving an average speedup of $129.88\times$. Notably, our average depth is $6.75\times$ smaller than that of `&transduction`, highlighting the effectiveness of our depth-aware heuristic during transduction.

C. Results of Transduction-Based Optimization Sequences

In this subsection, we compare transduction methods in a practical setting, where multiple optimization algorithms are iteratively applied as part of an optimization sequence to achieve optimal results. In the experiment, our optimization sequence interleaves our GPU-based transduction with existing GPU-based optimization sequences `resyn2`/`resyn2rs` from CULS. For comparison, we used the optimization sequence `&transtoch` as the baseline, as developed by the authors of [11]. This sequence interleaves `mfs2`, `if`, `dc2`, `strash` and `&transduction` in ABC with randomized parameters. Note that `&transtoch` and our sequence are not identical due to certain constraints. Specifically, the algorithms used by `&transtoch` do not have GPU versions, preventing

[1]https://www.iwls.org/iwls2022/
[2]https://github.com/cuhk-eda/CULS

TABLE II: Experimental Results (Running Time in Seconds)

Statistics				Single Transduction						Integrated Transduction					
				`&transduction` [11]			**Ours**			`&transtoch` [11]			**Ours** + CULS		
Case	#PIs	Size	Depth	Size	Depth	Time	Size	Depth	Time	Size	Depth	Time	Size	Depth	Time
ex48	16	1884	18	1507	265	437	1616	35	25	1385	168	606	1385	89	188
ex39	16	1944	19	954	101	144	1079	44	13	678	132	328	681	56	184
ex25	16	1949	20	922	117	184	1060	40	13	544	119	1392	549	46	186
ex30	16	2299	19	1242	197	786	1065	31	12	675	161	5451	675	41	184
ex26	17	3036	22	1166	175	302	1283	40	30	529	136	5530	546	40	186
ex27	18	4266	24	2275	247	2442	1480	39	64	432	128	45422	432	43	187
ex63	18	5512	22	5251	210	47144	5353	40	630	5311	87	17402	5311	27	632
ex65	18	10731	24	8936	793	285886	9036	43	1811	8885	130	231777	8887	35	1812
Average (Normalized)				1.01	6.75	129.88	**1.00**	**1.00**	**1.00**	1.00	2.81	86.54	**1.00**	**1.00**	**1.00**

us from using them. Additionally, while all GPU-based algorithms in our sequence have CPU implementations in ABC, they were not selected by the authors of `&transtoch` [11].

In this experiment, we ran and terminated our optimization sequence after approximately 3 minutes or after completing at least one full iteration of applying different algorithms. The `&transtoch` sequence was executed until it reached an AIG size comparable to ours. The results are presented in the "Integrated Transduction" column of Table II. As designed, both sequences yield similar size results. However, in terms of efficiency, our fully GPU-accelerated optimization sequence achieves an impressive speedup of over 86× compared to the CPU sequence. Additionally, our results demonstrate a 2.81× reduction in depth on average compared to the baseline.

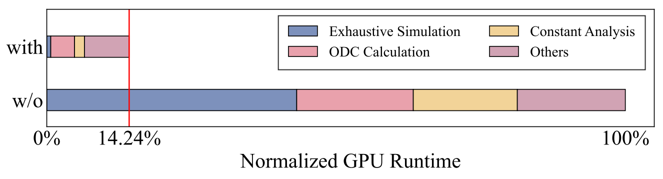

Fig. 5: GPU runtime with/without bit packing and related optimizations

TABLE III: Effectiveness of Individual Optimization

Configuration		Normalized Runtime	
Bit Packing	Optimization	ODC Calculation	Constant Analysis
×	×	1.00	1.00
✓	×	8.65	0.88
✓	#1	**0.28**	
✓	#2		**0.13**

D. *Effectiveness of Bit Packing and Related Optimizations*

This subsection will present the profiling results of our GPU-based transduction method (executed for one iteration only) using NVIDIA Nsight Systems on the benchmark `ex27`.

Overall Effectiveness. Figure 5 illustrates the distribution of runtime across our GPU kernels before and after the application of bit packing and two dedicated optimizations. The results show significant improvements in efficiency, with the total GPU runtime reduced by over 85%. The reductions are particularly noticeable in key processes such as exhaustive simulation and constant analysis, highlighting the overall effectiveness of our bit packing solution.

Effectiveness of Optimization #1. As discussed in Section IV-B, bit packing necessitates serialized modifications for bits packed in a 32-bit integer, which negatively affects GPU performance. Consequently, *ODC Calculation* experiences an 8.65× slowdown after bit packing, as depicted in Table III. By incorporating our optimization #1 to enhance thread utilization, the runtime is dramatically reduced by more than 30× (from 8.65 to 0.28), leading to an overall runtime reduction of 72% compared to the original version without bit packing.

Effectiveness of Optimization #2. As shown in Table III, the application of bit packing resulted in a 12% reduction in runtime for *Constant Analysis*. By implementing Optimization #2, as discussed in Section IV-C, we achieved further runtime reductions, showcasing the additional benefits of leveraging bitwise operations.

VI. Conclusion

In this paper, we present a parallel, GPU-accelerated transduction method. We begin by outlining our transduction workflow and introducing tailored and optimized parallelization strategies for each stage. Additionally, we propose a bit packing technique aimed at reducing GPU memory consumption. To tackle two efficiency challenges associated with bit packing, we developed two specific optimizations that significantly enhance performance. Extensive experimental results demonstrate a remarkable 130× speedup of our parallel transduction method compared to the state-of-the-art CPU-based approach.

VII. Acknowledgment

This work was partially supported by GDIC.

References

[1] 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*, 2006, pp. 532–535.

[2] A. T. Calvino and G. De Micheli, "Scalable Logic Rewriting Using Don't Cares," in *2024 Design, Automation & Test in Europe Conference Exhibition (DATE)*, 2024, pp. 1–6.

[3] H. Pan, K. Zhu, F. Yang, X. Zeng, S. Liu, Y. Xiao, Y. Shao, and Z. Chu, "Rethinking Logic Rewriting: Technology-Aware Subgraph Matching with Exact Synthesis," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 30, no. 5, Aug. 2025.

[4] H. Pan, Y. Xia, L. Wang, and Z. Chu, "Semi-Tensor Product-Based Exact Synthesis for Logic Rewriting," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 43, no. 4, pp. 1093–1106, 2024.

[5] C. Meng, W. Qian, and A. Mishchenko, "ALSRAC: Approximate Logic Synthesis by Resubstitution with Approximate Care Set," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, 2020, pp. 1–6.

[6] C. Meng, A. Mishchenko, W. Qian, and G. De Micheli, "Efficient Resubstitution-Based Approximate Logic Synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 44, no. 6, pp. 2040–2053, 2025.

[7] S.-Y. Lee, H. Riener, A. Mishchenko, R. K. Brayton, and G. De Micheli, "A Simulation-Guided Paradigm for Logic Synthesis and Verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 8, pp. 2573–2586, 2022.

[8] J. Kocnova and Z. Vasicek, "EA-Based Refactoring of Mapped Logic Circuits," in *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2019, pp. 1–5.

[9] R. Brayton and A. Mishchenko, "ABC: an academic industrial-strength verification tool," in *Proceedings of the 22nd International Conference on Computer Aided Verification*, ser. CAV'10. Berlin, Heidelberg: Springer-Verlag, 2010, p. 24–40.

[10] S. Muroga, Y. Kambayashi, H. Lai, and J. Culliney, "The transduction method-design of logic networks based on permissible functions," *IEEE Transactions on Computers*, vol. 38, no. 10, pp. 1404–1424, 1989.

[11] Y. Miyasaka, "Transduction Method for AIG Minimization," in *2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2024, pp. 398–403.

[12] L. Li, R. Li, and Y. Ha, "A Recursion and Lock Free GPU-Based Logic Rewriting Framework Exploiting Both Intranode and Internode Parallelism," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 42, no. 11, pp. 3972–3984, 2023.

[13] S. Lin, G. Guo, T.-W. Huang, W. Sheng, E. Young, and M. Wong, "GCS-Timer: GPU-Accelerated Current Source Model Based Static Timing Analysis," in *Proceedings of the 61st ACM/IEEE Design Automation Conference*, ser. DAC '24. New York, NY, USA: Association for Computing Machinery, 2024.

[14] L. Liu, B. Fu, S. Lin, J. Liu, E. F. Y. Young, and M. D. F. Wong, "Xplace: An extremely fast and extensible placement framework," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 43, no. 6, pp. 1872–1885, 2024.

[15] W. L. Lee, S. Jiang, D.-L. Lin, C. Chang, B. Zhang, Y.-H. Chung, U. Schlichtmann, T.-Y. Ho, and T.-W. Huang, "iG-kway: Incremental k-way Graph Partitioning on GPU," in *2025 62nd ACM/IEEE Design Automation Conference (DAC)*, 2025, pp. 1–7.

[16] Z. Wu, H. Zhao, H. Liu, W. Wen, and J. Li, "gHyPart: GPU-friendly End-to-End Hypergraph Partitioner," *ACM Trans. Archit. Code Optim.*, vol. 22, no. 1, Mar. 2025.

[17] B. Zhang, D.-L. Lin, C. Chang, C.-H. Chiu, B. Wang, W.-L. Lee, C.-C. Chang, D. Fang, and T.-W. Huang, "G-PASTA: GPU-Accelerated Partitioning Algorithm for Static Timing Analysis," in *Proceedings of the 61st ACM/IEEE Design Automation Conference*, ser. DAC '24. New York, NY, USA: Association for Computing Machinery, 2024.

[18] D.-L. Lin, Y. Zhang, H. Ren, B. Khailany, S.-H. Wang, and T.-W. Huang, "GenFuzz: GPU-accelerated Hardware Fuzzing using Genetic Algorithm with Multiple Inputs," in *2023 60th ACM/IEEE Design Automation Conference (DAC)*, 2023, pp. 1–6.

[19] S. Lin, L. Xiao, J. Liu, and E. F. Y. Young, "InstantGR: Scalable GPU Parallelization for Global Routing," in *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design*, ser. ICCAD '24. New York, NY, USA: Association for Computing Machinery, 2025.

[20] G. Guo, T.-W. Huang, and M. Wong, "Fast STA Graph Partitioning Framework for Multi-GPU Acceleration," in *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2023, pp. 1–6.

[21] G. Guo, T.-W. Huang, Y. Lin, and M. Wong, "GPU-accelerated Critical Path Generation with Path Constraints," in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2021, pp. 1–9.

[22] S. Lin, J. Liu, T. Liu, M. D. F. Wong, and E. F. Y. Young, "NovelRewrite: node-level parallel AIG rewriting," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, ser. DAC '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 427–432.

[23] Y. Sun, T. Liu, M. D. F. Wong, and E. F. Y. Young, "Massively Parallel AIG Resubstitution," in *Proceedings of the 61st ACM/IEEE Design Automation Conference*, ser. DAC '24. New York, NY, USA: Association for Computing Machinery, 2024.

[24] S. Lin, J. Liu, E. F. Y. Young, and M. D. F. Wong, "GAMER: GPU-Accelerated Maze Routing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 42, no. 2, pp. 583–593, 2023.

[25] J. Jiang, L. Zou, W. Zhao, Z. He, T. Chen, and B. Yu, "PDRC: Package Design Rule Checking via GPU-Accelerated Geometric Intersection Algorithms for Non-Manhattan Geometry," in *Proceedings of the 61st ACM/IEEE Design Automation Conference*, ser. DAC '24. New York, NY, USA: Association for Computing Machinery, 2024.

[26] C.-Y. Chiang, Z.-Y. Cai, C.-C. Lan, Y.-J. Chen, Y. Hsu, Y.-W. Chang, and H.-M. Chen, "Late Breaking Results: Scalable GPU-Friendly Parallelization for Sweep-Based Maze Routing," in *2025 62nd ACM/IEEE Design Automation Conference (DAC)*, 2025, pp. 1–2.

[27] Y. Zhang, H. Ren, and B. Khailany, "GL0AM: GPU Logic Simulation Using 0-Delay and Re-simulation Acceleration Method," in *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design*, ser. ICCAD '24. New York, NY, USA: Association for Computing Machinery, 2025.

[28] S. Lin and M. D. F. Wong, "Superfast Full-Scale GPU-Accelerated Global Routing," in *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, ser. ICCAD '22. New York, NY, USA: Association for Computing Machinery, 2022.

[29] L. Xiao, S. Lin, J. Liu, Q. Duan, T.-Y. Ho, and E. F. Y. Young, "InstantGR: Scalable GPU Parallelization for 3-D Global Routing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–1, 2025.

[30] H. Yang, K. Fung, Y. Zhao, Y. Lin, and B. Yu, "Mixed-Cell-Height Legalization on CPU-GPU Heterogeneous Systems," in *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2022, pp. 784–789.

[31] S. Liu, P. Liao, R. Zhang, Z. Chen, W. Lv, Y. Lin, and B. Yu, "FastGR: Global Routing on CPU-GPU with Heterogeneous Task Graph Scheduler," in *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2022, pp. 760–765.

[32] C. Zeng, F. Yang, and X. Zeng, "Accelerate Logic Re-simulation on GPU via Gate/Event Parallelism and State Compression," in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2021, pp. 1–8.

[33] A. B. Kahng and Z. Wang, "DG-RePlAce: A Dataflow-Driven GPU-Accelerated Analytical Global Placement Framework for Machine Learning Accelerators," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 44, no. 2, pp. 696–708, 2025.

[34] T. Liu and E. F. Young, "Rethinking AIG Resynthesis in Parallel," in *2023 60th ACM/IEEE Design Automation Conference (DAC)*, 2023, pp. 1–6.