GROOT: Graph Edge Re-growth and Partitioning for the Verification of Large Designs in Logic Synthesis

Kiran Thorat[1], Hongwu Peng[1], Yuebo Luo[3], Xi Xie[1], Shaoyi Huang[1], Amit Hasan[1],
Jiahui Zhao[1], Yingjie Li[2], Zhijie Shi[1], Cunxi Yu[2], Caiwen Ding[3]
[1]University of Connecticut [2]University of Maryland [3]University of Minnesota
[1]{kiran_gautam.thorat, hongwu.peng, xi.xie, shaoyi.huang, amit.hasan, jiahui.zhao, zhijie.shi}@uconn.edu
[2]{yingjiel, cunxiyu}@umd.edu [3]{luo00466, dingc}@umn.edu

Abstract*—Traditional verification methods in chip design are highly time-consuming and computationally demanding, especially for large scale circuits. Graph neural networks (GNNs) have gained popularity as a potential solution to improve verification efficiency. However, there lacks a joint framework that considers all chip design domain knowledge, graph theory, and GPU kernel designs. To address this challenge, we introduce GROOT, an algorithm and system co-design framework that contains chip design domain knowledge and redesigned GPU kernels, to improve verification efficiency. More specifically, we create node features utilizing the circuit node types and the polarity of the connections between the input edges to nodes in And-Inverter Graphs (AIGs). We utilize a graph partitioning algorithm to divide the large graphs into smaller sub-graphs for fast GPU processing and develop a graph edge re-growth algorithm to recover verification accuracy. We carefully profile the EDA graph workloads and observe the uniqueness of their polarized distribution of high degree (HD) nodes and low degree (LD) nodes. We redesign two GPU kernels (HD-kernel and LD-kernel), to fit the EDA graph learning workload on a single GPU. We compare the results with state-of-the-art (SOTA) methods: GAMORA, a GNN-based approach, and the traditional ABC framework. Results show that GROOT achieves a significant reduction in memory footprint (59.38 %), with high accuracy (99.96%) for a very large CSA multiplier, i.e. 1,024 bits with a batch size of 16, which consists of 134,103,040 nodes and 268,140,544 edges. We compare GROOT with GPU-based GPU Kernel designs SOTAs such as cuSPARSE, MergePath-SpMM, and GNNAdvisor. We achieve up to 1.104×, 5.796×, and 1.469× improvement in runtime, respectively.

I. Introduction

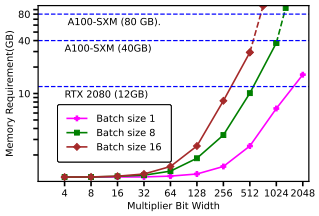
Logic synthesis plays a vital role in chip design by converting high-level circuit descriptions into optimized gate-level implementations and helps to bridge the gap between high-level synthesis and physical design [1]. Verification is a critical step in logic synthesis that ensures internal functionality, prevents costly errors, and reduces the time-to-market by identifying and fixing issues early in the design cycle [2]. However, traditional verification methods are computationally demanding and increasingly time-consuming, especially for complex designs [3], [4]. For example, as measured in [5], the verification process takes more than 100 hours for the booth multiplier using the OneSpin commercial equivalence checker tool. Furthermore, using the open-source verification tool ABC [6], a 2048-bit multiplier requires 8.6×10^5 seconds (more than nine days) [7].

Graph neural networks (GNNs) have gained popularity as a potential solution to improving verification efficiency, e.g., GAMORA [7], since graph is one the natural ways to represent many fundamental objects in circuits, such as Register Transfer Level (RTL) descriptions, netlists, layout, and Boolean functions. In GNN-based methods, GNN is leveraged to classify the graph nodes which significantly reduces the verification time. For example, the 2048-bit multiplier verification time reduces from more than nine days to 0.919 seconds when GNN is used [7].

Despite their promising results, there are research gaps. **First**, an effective graph machine-learning solution for logic synthesis requires a fusion of electronic design automation (EDA) domain expertise and knowledge of graph machine learning. However, existing efforts tend to focus on just one aspect, such as applying GNN algorithms to EDA tasks, and may lack EDA domain expertise. For instance, GAMORA [7] does not distinguish Primary Inputs and Primary Outputs (PO) when creating graph node features, however, PI and PO are inherently different and need to be distinguished. **Second**, processing a large-scale EDA GNN on a single hardware, which is crucial to efficient AI, has been largely neglected. Figure 1 (a) shows the memory consumption (on two high-end GPUs NVIDIA A100 40 GB and 80 GB, and one low-end GPU GeForce RTX2080) required for the verification of various bit widths multipliers. We observe that even the NVIDIA A100 could not accommodate the 1,024-bit CSA multiplier graph when batch size equals 16. Please note that batch processing is essential to achieve high throughput as GPUs are designed to process parallel data. **Third**, the state-of-the-art (SOTA) high-performance solutions often use GPU, and simply adopt commercialized multi-GPU solutions (e.g., GAMORA directly uses Pytorch Geometric [8] on two or more GPUs). However, an important aspect that frequently goes unnoticed is the consideration of GPU primitives. This fundamentally limits making single GPU achievable for EDA GNN and the applicability of broadening accessibility in economically disadvantaged districts.

In this research, we propose GROOT, Graph Edge Re-growth and Partitioning for the Verification of Large Designs in Logic Synthesis. GROOT is a single-GPU-based framework

This work was supported in part by the National Science Foundation under IUCRC-1916762, and CHEST (Center for Hardware Embedded System Security and Trust) industry funding.



(a)

Methods	Large multipliers	Run time	Accuracy
ABC [6]	✓	×	✓
GAMORA [7]	×	✓	✓
GROOT (ours)	✓	✓	✓

(b)

Fig. 1: (a) Extremely high GPU memory requirements on large circuit graphs in EDA. CSA multiplier with different bits and batch sizes, (b) Comparison of verification methods.

and simultaneously achieves high accuracy, and low memory footprint at run-time. The classical open-source EDA tool ABC [6] is not capable of obtaining verification results at run-time, and GAMORA [7] faces the out of memory issue on large circuit graphs, as summarized in figure 1 (b).

Our key contributions are:

- At the EDA domain level, we revisit and redesign the node features for the EDA graphs and create datasets. We use circuit node types and the polarity of connections between input edges and nodes in And-Inverter Graphs (AIGs) to form the input embedding. Adding more features to the EDA graphs helps the GNN model learn broader characteristics of designs.
- At the graph processing level, we use EDA domain knowledge and a graph partitioning algorithm to split large graphs into smaller sub-graphs for GPU processing. Further, we develop a boundary edge re-growth algorithm for accuracy recovery.
- We carefully profile the EDA graph workloads and note the polarized distribution of high degree (HD) nodes and low degree (LD) nodes. We redesign two GPU kernels (HD-kernel and LD-kernel) to accelerate the EDA graph learning workload on a single GPU.

Experimental results show that GROOT achieves a significant reduction in memory footprint (59.38 %), with high accuracy (99.96%) for large circuit graphs, compared to ABC and GAMORA. Note that the accuracy of node classification directly translates to the verification accuracy. Compared to SOTA GPU-based GPU Kernel designs such as cuSPARSE [9], MergePath-SpMM [10], and GNNAdvisor [11], GROOT achieves up to 1.104×, 5.872×, and 1.469× improvement in runtime, respectively.

II. Background and Related Work

Verification in EDA Verification can be performed at multiple stages to ensure that the designed chip meets its intended functionality.

Traditional formal verification techniques include Satisfiability (SAT), canonical diagrams, theorem proving [12], and algebraic re-writing. The SAT technique models the verification problem as Boolean satisfiability [13], [14]. Canonical diagrams propose different graph-based representations, such as binary decision diagrams (BDDs) [15], Taylor expansion diagrams (TEDs) [16], and binary moment diagrams (BMDs) [17]. The algebraic approaches, based on modeling circuit specifications and hardware implementation as polynomials [18], leverage symbolic computer algebra techniques [14], [19], [20] to solve verification problems.

Multiplier Case Study Arithmetic circuits, especially those with large integer multipliers, play a key role in many areas. They are used in homomorphic encryption [21], security tasks such as malicious hardware detection [22], multimedia processing [4], and signal processing [4]. We focus on multipliers because they belong to the challenging combinational arithmetic designs and showing our framework on multipliers suggests it is suitable for simpler blocks such as full adders [4], [7]. However, verifying these circuits, particularly the large integer multipliers remains a challenge [3], [12], [14]. Large integer multipliers are also critical for tasks such as scientific computing [23], financial operations [24], and cryptography [25]. One approach uses Symbolic Computer Algebra (SCA) to verify different multipliers by recognizing full adders (`FAs`) and half adders (`HAs`) in the netlists. For example, [5] shows that a 128-bit multiplier based on ripple carry adders takes more than one hour to verify, while a Booth multiplier takes over 100 hours when using the OneSpin commercial equivalence checker tool.

GNN in EDA Graph neural networks (GNNs) learn graph structures and extract useful information [26]. In EDA, circuit netlists naturally form graphs, which makes GNNs a good fit for this domain. For example, NeuroSAT [27] uses message passing in a neural network to learn SAT problems and predict satisfiability. In another study [28], GNNs predict testability for netlists with performance similar to commercial tools. These ML and GNN techniques show promise for verification, but further work is needed to improve their performance and to address challenges in scalability and data management [7].

III. GNN for Verification in Logic Synthesis

The overview of GROOT framework is depicted in Figure 2, consisting of five stages, i.e., (a) Converting the netlist into a transitional graph representation using an open-source EDA tool ABC [6]; (b) Pre-process the transitional graph and generate the standardized logic synthesis-based EDA graph; (c) Partition of the large EDA graphs; (d) Utilize GNN for aggregation and message passing; and (e) Node classification and post-processing.

A. Converting Netlists into Transitional Graph

A Boolean network (digital design) can be described as a directed acyclic graph (DAG), where the nodes symbolize logic gates and edges symbolize connecting wires. An And-Inverter Graph (AIG) represents a specific type of combinational Boolean network, comprised of two input `AND` gates and `inverters` [29]. Essentially, AIG graphs are specialized DAGs that encapsulate the logical functionality of Boolean networks. Interestingly, through DeMorgan's rule, the combi-

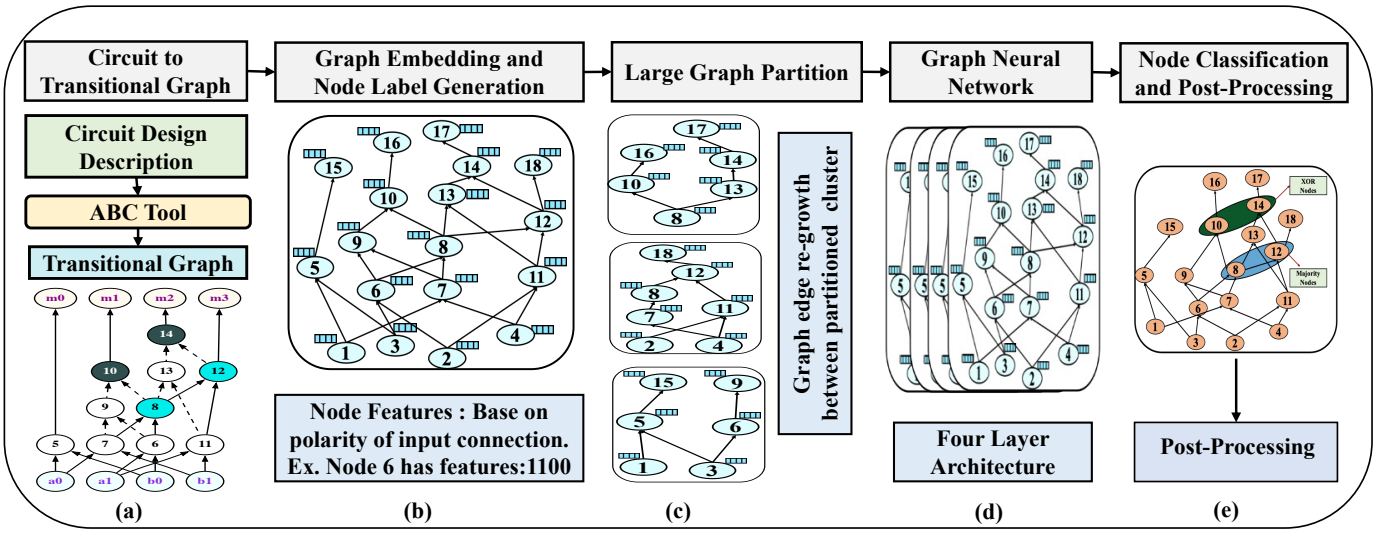


Fig. 2: Overview of the tasks: (a) Circuit to Transitional Graph Conversion, (b) Graph Input Embedding and Node Label generation based on transitional graphs, (c) Large Graph Partitions to Solve GPU memory issues, (d) Graph Neural Network Architecture, (e) Node classification.

national logic of any given Boolean network can be easily transformed into an AIG.

In GROOT, this transformation is accomplished through an open-source EDA tool ABC [6]. Figure 3 illustrates the transformation process using a two-bit CSA multiplier. The ABC takes a netlist as an input, as shown in Figure 3 (a), and generates the corresponding AIG representation, shown in Figure 3 (b). In AIG representation, inputs $a1a0$ and $b1b0$ represent the two-bit binary numbers for the multiplier and multiplicand, respectively. The multiplication result is represented using $m3m2m1m0$ bits. For example, multiplier $a1a0 = 10$ and multiplicand $b1b0 = 11$ gives the multiplication result $m3m2m1m0 = 0110$. The multiplication of the least significant bits (LSB) highlighted in golden color, symbolizing the '`AND`' operation at node 5 (i.e., $m0 = a0 \cdot b0$). The additional operation of multiplication is '`XOR`' (green), performed at nodes 6, 7, 8, 9, and 10, and can be represented by the equation $m1 = a0 \cdot b1 \; xor \; a1 \cdot b0$. The '`NOT`' operations are indicated by dashed lines.

B. Node Features and Node Label creation

We take the node and edge information from the AIG representation (transitional graph) to form the EDA graph. We define circuit-based EDA graph as $G = (V, E)$ with N nodes $v_i \in V$ and edges $(v_i, v_j) \in E$. We use an adjacency matrix $A \in R^{N \times N}$ to describe graph connections, a degree matrix $D_{ii} = \sum_j A_{ij}$ and a feature matrix $X = \{x_1, x_1, ..., x_N\}$.

We transform an AIG graphs (Figure 3, (d)) into graph embedding. We create node features by explicitly encoding two aspects: node type and input edge polarity. Each node in the AIG belongs to one of three categories Primary Inputs (`PI`), Internal logic gates (`AND` nodes), or Primary Outputs (`PO`) and is represented by a 4-bit vector as shown in figure 3, (c).

First two bits (node type) are encoded as:'00' for Primary Input (`PI`), '11' for Internal logic node (`AND` gate), '0X' for Primary Output (`PO`) where 'X' indicates polarity inherited from the preceding internal node. Last two bits (input polarity for internal nodes) are encoded as: '00' for Both inputs non-inverted '01' or '10' for Exactly one input inverted ('01' for right, ''10' for left input) '11' for Both inputs inverted. `PI` nodes have no incoming edges, thus their polarity bits are always '00'. `PO` nodes inherit polarity information from preceding internal nodes. For instance (Figure 3, (c) vector table), node 5, an internal node with non-inverted input edges, has a feature vector of 1100. Similarly, node 10, another internal node with inverted inputs, has a feature vector of 1111. The `PI` (node 1) or $a0$ has a feature vector of 0000 since no input connections, while the `PO` node 15 or $m0$ has a feature vector of 0011 since it inherited non inverted input as highlighted in red dotted lines between the figures 3 (c) and 3 (f). Our EDA graph embedding consist of four-node features, a distinction from the three-node features in GAMORA [7]. Implementing additional node features offers a more robust representation of nodes and improved generalization.

Next, we create labels for the ground truth using ABC [6]. Figure 3 (c), depicts the labels for the two-bit CSA multiplier. For nodes 1 to 4 (`PI` nodes), we label them as 4. For nodes 5, 6, 7, 9, 11, 13 (two-input `AND` gates), we label as 3. For nodes 10 and 14 (`XOR`), we label as 2. For nodes 12 and 8 (`MAJ` functionality) are labeled as 1. Lastly, all `PO` nodes, namely 15 to 18, are labeled as 0.

C. GNN and Edge Re-growth After Partition

To handle the memory footprint of large EDA graphs, we partition the graph into subgraphs as shown in Figure 2 (c) and feed them to a GNN for node classification. We use GraphSAGE [30]. The layer takes a graph with node features

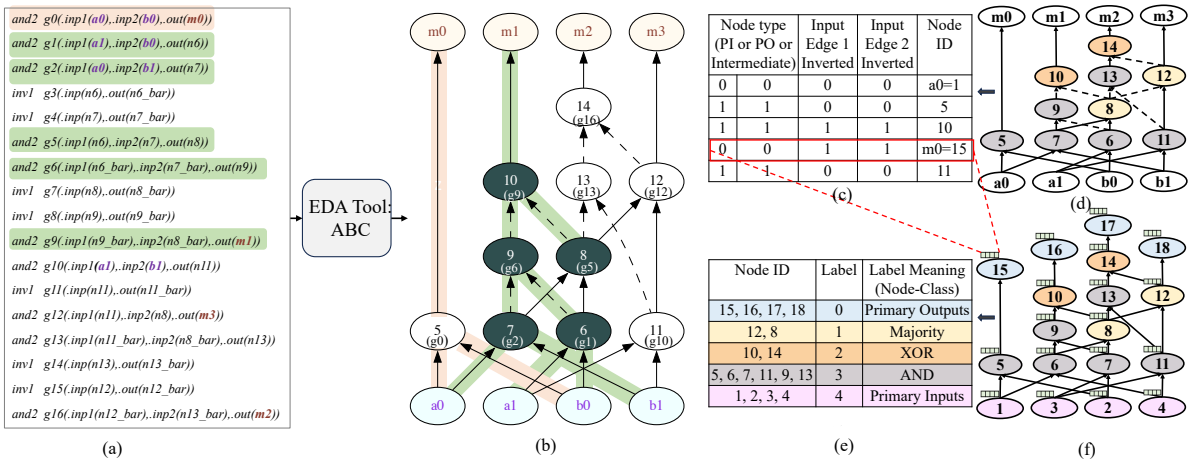


Fig. 3: Input to EDA graph flow: (a) Two-bit multiplier netlist, (b) AIG representation of two-bit multiplier using ABC (the dotted line represents inverted inputs to node), (c) Node features (selected nodes shown), (d) EDA graph of two-bit CSA multiplier, (e) Ground truth labels for the GNN model. (f) EDA graph embedding with node features of two-bit CSA multiplier.

as input (Figure 2 (d)) and aggregates features. We then apply METIS [31] to partition the graph.

Let p index a partition and let S_p be its node set. For a node u, $\mathcal{N}(u)$ is the set of nodes one hop from u. For edges, $(i, j) \in E$ denotes a directed edge from node i to node j, and $E[S]$ denotes the edges whose endpoints are both in S.

For partition p, we collect one-hop neighbors and the boundary nodes:

$$N(S_p) = \bigcup_{u \in S_p} \mathcal{N}(u), \qquad B_p = N(S_p) \setminus S_p, \tag{1}$$

where $N(S_p)$ is the set of all nodes that are exactly one hop away from any node in S_p, and B_p keeps only those neighbors that are outside S_p (the boundary nodes of partition p).

Next, we define the edges that cross between S_p and its boundary B_p, and the augmented sets:

$$\begin{aligned} C_p &= \{\, (i,j) \in E \mid (i \in S_p \wedge j \in B_p) \ \vee \ (i \in B_p \wedge j \in S_p) \,\}, \\ S_p^+ &= S_p \cup B_p, \\ E_p^+ &= E[S_p] \cup C_p. \end{aligned} \tag{2}$$

Here, C_p contains all edges with one endpoint in S_p and the other in B_p; S_p^+ and E_p^+ are the node and edge sets after adding these boundary nodes and crossing edges. Algorithm 1 performs boundary re-growth per partition: it first identifies the boundary nodes B_p using Eq. (1), then collects all crossing edges C_p using Eq. (2), and finally forms the augmented node and edge sets.

We observe that EDA graphs contain approximately 10% boundary edges between partitions, and the boundary recovery process does not add significant complexity to the inference stage. This approach regenerates boundary edges between disconnected partitions to prevent the loss of features and supports message passing between inter-partition nodes.

Algorithm 1 Graph Boundary Edge Re-growth

Require: $G = (V, E)$; partitions $\{S_p\}_{p=0}^{n}$

1: **for** $p = 0$ **to** n **do**
2: $B_p \leftarrow \left(\bigcup_{u \in S_p} \mathcal{N}(u)\right) \setminus S_p$ {boundary nodes; Eq. (1)}
3: $C_p \leftarrow \{(i,j) \in E : (i \in S_p \wedge j \in B_p) \ \vee \ (i \in B_p \wedge j \in S_p)\}$ {crossing edges; Eq. (2)}
4: $S_p^+ \leftarrow S_p \cup B_p$; $E_p^+ \leftarrow E[S_p] \cup C_p$ {augmented sets; Eq. (2)}
5: **end for**
6: **return** $\{(S_p^+, E_p^+)\}_{p=0}^{n}$

D. Node Classification and Post-Processing

We use GNN to classify the nodes into two categories `XOR` and `MAJ` as depicted in Figure 2 (e). We use the algebraic re-writing technique [4], [20] for verification. The algebraic representation of the basic Boolean operators is summarized in the Table I. Consider the case for `XOR3` and `MAJ` operations.

TABLE I: Algebraic Representations of Basic Boolean Operators (a, b, c are inputs)

Operation	Boolean Model	Algebraic Model
NOT	$\neg a$	$1 - a$
AND	$a \wedge b$	ab
XOR	$a \oplus b$	$a + b - 2ab$
XOR3	$a \oplus b \oplus c$	$a + b + c - 2ab - 2ac - 2bc + 4abc$
MAJ	$(a \vee b) \wedge (a \vee c)$	$ab + ac + bc - 2abc$

The sub-polynomial expression is $x_1 + 2x_2 + \ldots$, where $x_1 = \texttt{XOR3}(a, b, c)$ and $x_2 = \texttt{MAJ}(a, b, c)$, where a, b, c are inputs of `XOR3` and `MAJ` functions. Substituting the algebraic representations of `XOR3` and `MAJ` into the sub-polynomial, we

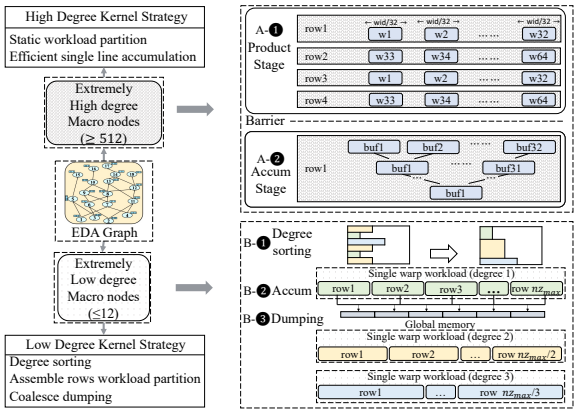


Fig. 4: GPU Kernel Design for EDA.

obtain:

$$\begin{aligned} x_1 + 2x_2 + \ldots &= (a + b + c - 2ab - 2ac - 2bc + 4abc) \\ &\quad + 2(ab + ac + bc - 2abc) \\ &= a + b + c \end{aligned}$$

This simplification results in the elimination of the four nonlinear terms: $2ab$, $2ac$, $2bc$, and $4abc$.

This polynomial reduction based on algebraic re-writing [4], [20] and is integrated in ABC [6]. This approach is reliant on detecting `XOR` and `MAJ` gates from a flattened netlist, a process that tends to be time-consuming. We leverage the GNN node classification to detect `XOR` and `MAJ` gates which makes verification efficient. As depicted in Figure 2 (e), in the two-bit CSA multiplier, nodes 10 and 14 are classified as `XOR`, while nodes 12 and 8 are classified as `MAJ`. These nodes are used for verification using the method described in [4]. Note that the accuracy of node classification directly translates to the verification accuracy. In the following section, we describe kernel design which is necessary to train GNN with larger multipliers. More details are in subsection V-A Accuracy Enhancement with Large Design Training part.

IV. Kernel Design - GROOT-GPU

We tailor GPU kernels (high-degree (HD) kernel and low-degree (LD) kernel) separately for the extremely high-degree macro nodes (≥ 512) and the low-degree macro nodes (≤ 12). The whole GPU kernel is programmed in CUDA C. We start by partitioning the workload (non-zero elements) statically for each row of the adjacency (A) matrix (all nodes possessing a degree equal to the width). We show an example in the Fig. 4. The HD macro nodes contain 4 rows, namely $row1$ to $row4$. Each row contains wid non-zero elements. Each block in the kernel contains 64 warps, numbered from $w1$ to $w64$. We divide each row into 32 equal workloads, each containing $\frac{wid}{32}$ non-zero elements. Then we assign the workloads in $row1$ to the warps numbered $w1$ to $w32$ in turn, and assign the workloads in $row2$ to warps from $w33$ to $w64$.

The LD-kernel design for low-degree macro nodes is shown in the lower half of Fig. 4. Step B processes the degree sorting on the adjacent matrix with the following steps: (1) computing each row's degree using the row pointer array with time complexity of $\mathcal{O}(n)$ when employing count sort [32] or radix sort [33], with n indicating the number of rows; (2) applies a stable sorting algorithm to sort rows by their degrees; and (3) updating the row pointer array to reflect the new rows' order, with time complexity of $\mathcal{O}(n)$. The dominant time complexity of this operation arises from applying the stable sorting algorithm. Nevertheless, employing count sort, a linear time-complexity algorithm, can optimize the overall time complexity to $\mathcal{O}(n)$. The details of sorting and row-assembling are as described in Figure 5. The warp-block operation of LD-kernel starts with sorting on the original sparse input by the degree of each row. It maps the rows into an array linearly, in which the partitioning is executed by dividing the array into sections of rows according to their degree, sequentially from the smallest to the largest. Then, rows with the same degree in every section are further partitioned into blocks, whose warps will be processed in parallel to extract rows and multiply them with the corresponding right-hand-side (RHS) rows from the dense input. The resulting product rows sum up by the degree of the left-hand side rows to produce the output rows. This blockwise operation is performed in parallel with other blocks. For example, in the lower left part of the figure, warp 1 traverses its non-zeros one by one from the left to the right, while locating the corresponding RHS row (1,1). The first 1 implies the warp 1, the second 1 refers to the first row, which warp 1 is responsible for. Then, the first traversed non-zero from warp 1 multiplies the RHS row (1,1), resulting in the output RHS row 1 in the right part of the figure, and so forth for the rest of warp 1 and other warps. Multiple rows of non-zero elements are assigned to the same warp, rather than the maximum number of non-zero elements from one row per warp. Since the degree of each row is small, say 3, the number of warps per block in this kernel is set to $6m$. For rows with degrees of 1, 2, and 3, each warp is responsible for $6m$, $3m$, and $2m$ rows, which translates into nz_{max}, $nz_{max}/2$ and $nz_{max}/3$ rows in the figure, respectively. This achieves workload balance between warps and helps to increase the proportion of active warps. By aggregating the workload of many small degree rows into the same warp, we will significantly reduce the overhead of warp switching and improve calculation efficiency compared to the ordinary SPMM algorithm process. Moreover, this aggregated organization method will further enhance access continuity to global memory when transferring calculation results from shared memory to global memory. We can utilize the technique of coalesce dumping to exploit this advantage fully. During step $B-$❸, termed coalesce dumping, we will write the intermediate results of multiple consecutive rows, handled by the same warp, into a continuous area in global memory. This approach will significantly enhance the efficiency of accessing global memory.

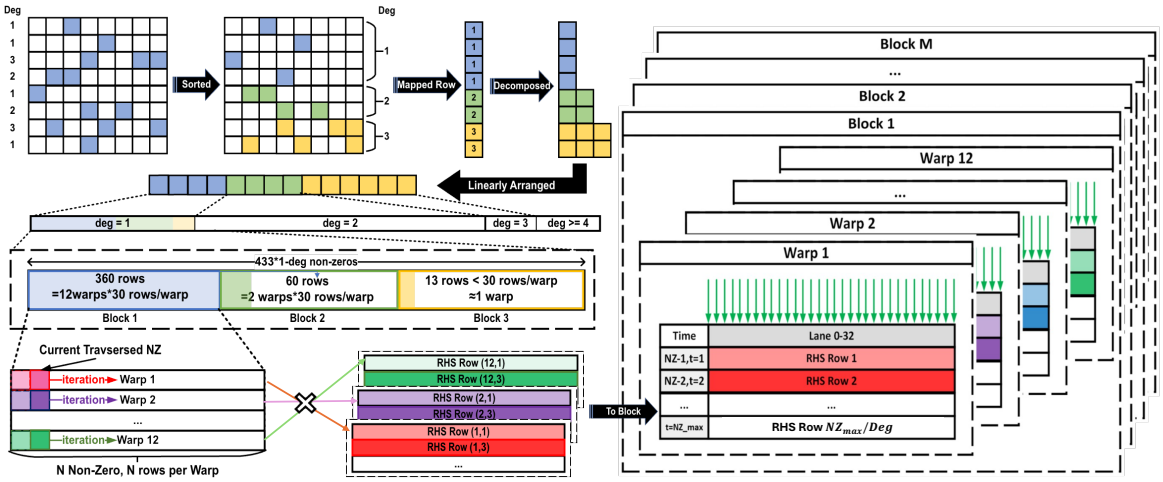


Fig. 5: Detailed process of LD-kernel, from degree-sorting, row-assembling, block-partitioning, to warp-wise multiplication and summation, with block-wise parallelism.

V. Performance Evaluation

This section presents the evaluation of GROOT by comparing its memory usage, and run-time against two baselines: the traditional open source tool ABC [6] and the GNN-based SOTA GAMORA [7]. When comparing with [7], we run the GAMORA framework against our modified datasets. We use a Linux-based host with AMD EPYC 7543 32-Core Processor and an NVIDIA A100-SXM 80 GB.

A. Accuracy and Accuracy Recovery of Various Multipliers

Our GNN model is trained on an 8-bit multiplier and then used in inference on larger multipliers of the same dataset. Figure 6 depicts the accuracy with respect to the number of partitions for different datasets. Figure 6 (a) shows the accuracy on CSA multipliers with batch size one. Without any partitioning, we achieve high accuracy, reaching 100% for multipliers of sizes 128-bits and above, while the accuracy is 99.94% for the 32-bit multiplier. As the number of partitions increases, the loss in accuracy becomes more noticeable because more partitions require the removal of more boundary edges. However, using our boundary edge re-growth approach effectively recovers accuracy. In Figure 6 (a), the solid line denotes the regained accuracy when using boundary edge re-growth. Notably, this edge re-growth method achieves a maximum recovery of 8.7% boost in accuracy of a 32-bit multiplier. By adopting our edge re-growth approach, one can afford to use more partitions while maintaining high accuracy To evaluate the scalability of GROOT, we evaluate its accuracy on large CSA multipliers such as the 1024-bit multiplier with a batch size of 16 containing 134,103,040 nodes and 268,140,544 edges. The figure 6 (b) shows accuracy with respect to number of partions. The trends show that the accuracy is at 100% up until 16 partitions. This accuracy can be attributed to the presence of a large number of edges in these large graphs and a small number of edges removal does not affect message passing in GNN. Consequently, Partitioning does not much impact the accuracy. However, post the 16-partition mark, there is a slight drop in accuracy since more edges are removed to create partitions.

To evaluate GROOT's performance with complex graphs, we utilize the Booth multipliers dataset. Figure 6 (c) shows accuracy with respect to the number of partitions. The accuracy drop is more compared to that in other datasets. However, the utilization of the edge re-growth approach enables the mitigation of this accuracy drop, as illustrated by the solid line in Figure 6 (c). The re-growth achieves a maximum 12.62% accuracy recovery in a 32-bit multiplier. Additionally, we test with the ASAP 7nm technology [34] mapped netlist dataset. This netlist comprises 161 standard cell gates, including the multi-output gate, leading to certain irregularities. As evidenced in Figure 6 (d), even with such irregularity, GROOT shows high accuracy and maintains more than 76% accuracy after edges re-growth. In summation, GROOT is capable of handling design complexities.

Figure 7 (a) shows the accuracy of FPGA-mapped CSA multipliers with batch size equal to 1 and the model is trained on 8 bits. The accuracy is low among all the datasets. To further improve prediction accuracy, we focus on training the GNN model using larger multipliers. This approach significantly improves the model's prediction accuracy. When the model trained on a 64-bit multiplier boosts the accuracy for a 64-bit multiplier from 71.82% (Figure 7 (a), number of partition=1) to 90.8% (Figure 7 (b), number of partition=1) an 18.98% boost in accuracy. However, this accuracy gain comes with increased training time. Training a 64-bit FPGA for 100 epochs takes 2914.42 seconds. To mitigate this time cost, we propose designing a specialized kernel for faster matrix multiplication, a major factor in training and inference time.

B. Memory Footprint Analysis

Figure 8 illustrates the GPU memory utilization by various multipliers with respect to the number of partitions. Figure

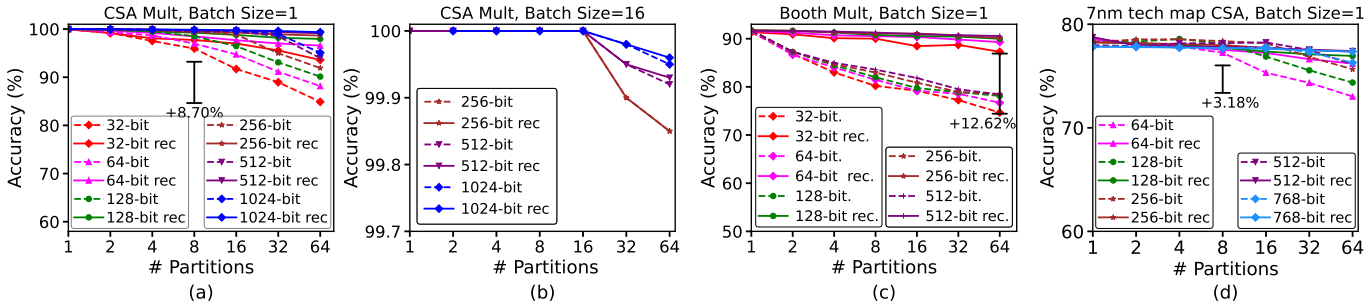


Fig. 6: Verification accuracy as a function number of partitions for (a, b) CSA multipliers, batch size 1 and 16, respectively; (c) Booth multipliers, batch size 1; (d) CSA multipliers after 7nm technology mapping, batch size 1. All the multipliers were trained using 8-bits.

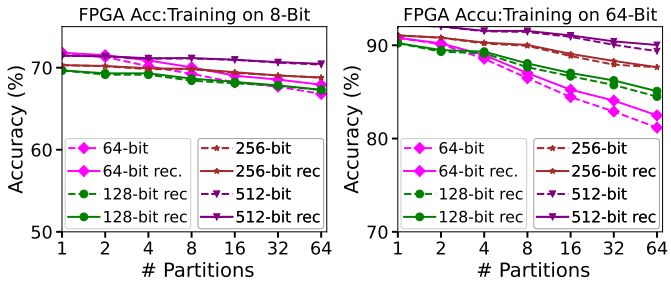


Fig. 7: FPGA multiplier dataset accuracy improvement (a) 8-bit training, (b) 64-bit training

TABLE II: Large Multiplier GPU Memory Usage (In MB) Comparison. (Batch size=16, OOM= Out of Memory).

# Part.	256-Bit	512 -Bit	1,024-Bit
GAMORA [7]	8,263	29,375	**OOM**
GROOT 2 Part.	5,457	18,135	68,923
GROOT 4 Part.	3,923	13,025	48,463
GROOT 8 Part.	3,157	8,421	32,093
GROOT 16 Part.	2,901	7,909	27,997
GROOT 32 Part.	2,901	7,909	27,997
GROOT 64 Part.	2,901	7,909	27,997

8 (a) shows the GPU memory utilization on the y-axis and the number of partitions on the x-axis for CSA multipliers with batch size one. As the number of partitions increases, the memory requirement decreases. For larger multipliers (e.g., 1024 bits), the memory reduction trend follows an exponential decay. When partitioned into 64 sub-graphs, the 1,024-bit multiplier showed a maximum benefit of 64.94% reduction in memory requirement as per depicted in the Figure 8 (a).

To evaluate the scalability of GROOT, we evaluate its performance on massive multiplier graphs such as the 1024-bit multiplier with a batch size of 16 containing 134,103,040 nodes and 268,140,544 edges, as depicted in Figure 8 (b). Partitioning the 1,024-bit multiplier into 64 sub-graphs resulted in a maximum memory reduction of 59.38%. Without partitioning, even high-end GPUs such as NVIDIA A100-SXM with 80 GB memory cannot perform verification on this massive graph. Thus, GROOT offers a fundamental solution to scalability. Our method is different from GAMORA [7], which requires multiple GPUs to handle massive graphs while we only need one low-end GPU. Table II shows the different multipliers and their GPU memory utilization.

Furthermore, to recover the accuracy, our algorithm regrows the edges after partitioning. The effect of the number of partitions on the memory requirement can be observed until the number of partitions is equal to 16. When the partitioning size is large (say 32) as shown in Figure 8 (b), the recovered edge consumes a large portion of the memory footprint, thus we observe less memory saving.

To demonstrate GROOT's effectiveness on complex designs, we evaluate it on different complex datasets. Figure 8 (c) shows memory utilization versus the number of partitions for booth multipliers, indicating an exponential reduction in memory with respect to partitions. The 512-bit booth multiplier shows maximum memory requirement reduction which is 41.84%. Figure 8 (d), shows memory utilization versus number partitions for 7nm mapped CSA multipliers, demonstrating a significant reduction in memory requirement for post-technology-mapped CSA multipliers. For instance, the maximum memory requirement reduction for the 768-bit multiplier is 70.15%. Similarly, Figure 7 (c) shows memory utilization for FPGA-mapped CSA multipliers. The maximum memory requirement reduction is 57.62% for a 512-bit multiplier. The benefits of memory reduction remain with increased design complexity.

C. Run Time Analysis and Comparative Study

Figure 10 shows the inference run time of verifying different multipliers of different widths after applying boundary edge re-growth for accuracy recovery. As shown in Figure 10 (a), as the bit width increases, ABC's run time expands exponentially compared to both GROOT and GAMORA. In comparison, GROOT significantly outperforms ABC [6]. For instance, when processing graphs for 1,024-bit CSA multipliers partitioned into 64 subgraphs, GROOT achieves a speedup of 1.23×10^5 over ABC. Additionally, the verification times for partitioned graphs using GROOT closely align with GAMORA [7]. It is important to note that the verification time for GROOT depends on the number of partitions, with

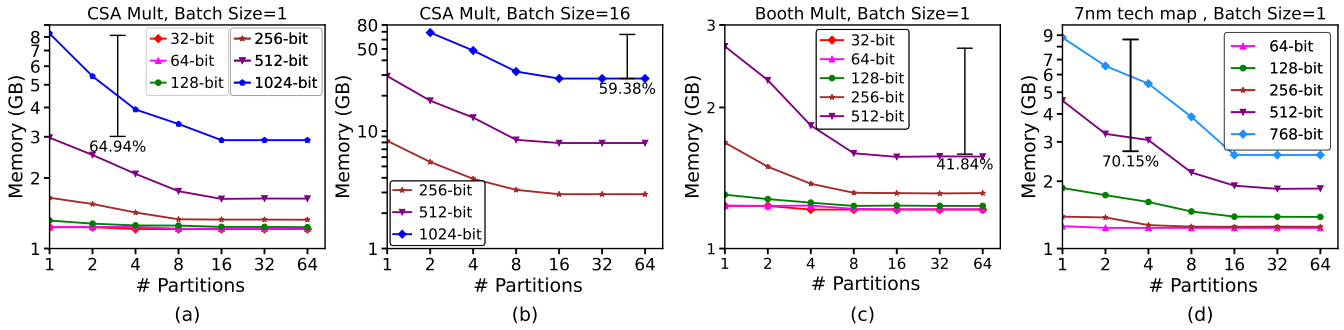


Fig. 8: Memory utilization as a function of the number of partitions for (a) CSA multipliers, for a batch size of 1, (b) CSA multipliers, for a batch size of 16, (c) Booth multipliers, for a batch size of 1, and (d) CSA multipliers, following the application of 7nm technology mapping, with a batch size of 1.

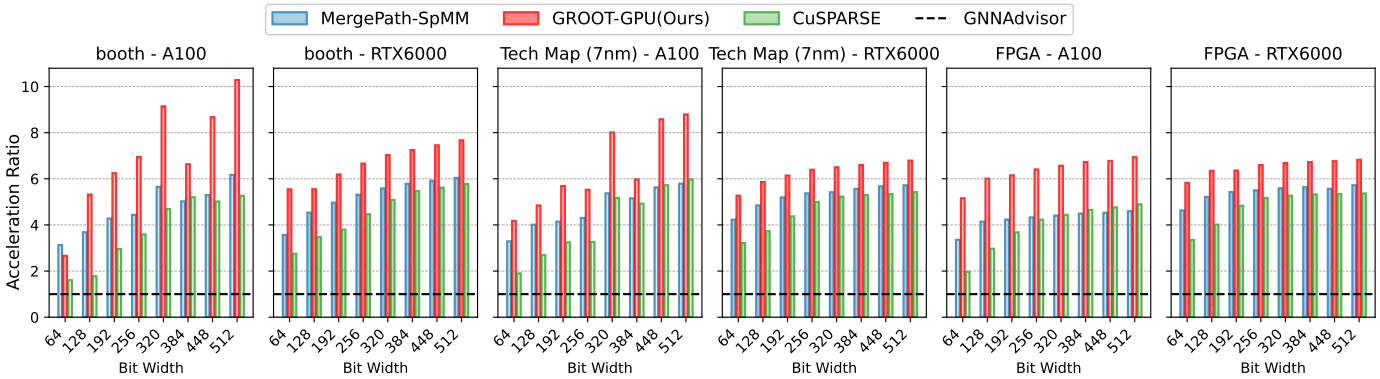


Fig. 9: The runtime comparison among GROOT-GPU and SOTA GPU-based GPU Kernel designs, where the acceleration ratio of 1 from GNNAdvisor is drawn as the black dash line.

more partitions slightly increasing the verification time due to additional partitioning overhead. However, neither GAMORA nor ABC can efficiently handle large graph datasets on a single GPU, as shown in Figure 1.

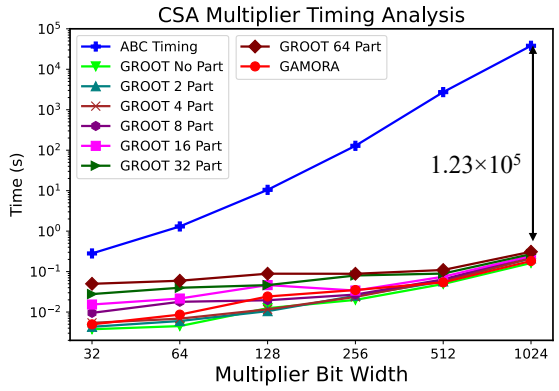


Fig. 10: CSA multiplier verification time comparisons with ABC [6] and GAMORA [7].

D. GPU kernel results

We compare our design with SOTA GPU-based GPU Kernel designs such as cuSPARSE [9], MergePath-SpMM [10], GNNAdvisor [11]. Figure 9 shows the comparison results of MergePath-SpMM, our GROOT-GPU, and CuSPARSE against GNNAdvisor, represented by the black horizontal dashed line. The kernels are tested on the graph of the Booth Multiplier, Technology Mapping, and FPGA 4LUT datasets with bit widths ranging from 64 to 512 and an embedding dimension of 32. The kernels perform SpMM operations given the graph adjacency matrices with corresponding embeddings, and the runtime of SpMM operations are recorded by the type of kernels, the bit width of the net list which graphs describe, and the datasets where the graphs belong to. Our GROOT-GPU demonstrates superior acceleration compared to the other three SOTA SpMM kernels in most cases. The performance gap widens as the bit width of the multiplier datasets increases and with more powerful GPUs. GROOT-GPU achieves the highest acceleration ratio of 10.28 for the Booth dataset with a bit width of 512 on the A100 GPU, outperforming the second-fastest MergePath-SpMM by 1.67× and the third-fastest CuSPARSE by 1.95×.

VI. Conclusion

In this paper, we introduce GROOT, an algorithm and system co-design framework that contains chip design domain knowledge, graph theory, and redesigned GPU kernels, to improve verification efficiency. We redesign nodes features utilizing the circuit node types and the polarity of the con-

nections between the input edges to nodes in And-Inverter Graphs (AIGs). We utilize a graph partitioning algorithm to divide the large graphs into smaller sub-graphs for fast GPU processing. After profiling EDA graph workloads, we notice their distinct distribution of high-degree and low-degree nodes and tailor the GPU kernel accordingly.

We compare the results with state-of-the-arts, e.g., GAMORA and ABC. Experimental results show that GROOT achieves a significant reduction in memory footprint, with high accuracy for a very large CSA multiplier, i.e., 1,024 bits with a batch size of 16. We also compare GROOT with SOTA GPU-based GPU Kernel designs such as cuSPARSE, MergePath-SpMM, and GNNAdvisor, and achieve notable runtime improvement.

REFERENCES

[1] J.-H. R. Jiang and S. Devadas, "Chapter 6 - logic synthesis in a nutshell," in *Electronic Design Automation*, L.-T. Wang, Y.-W. Chang, and K.-T. T. Cheng, Eds. Boston: Morgan Kaufmann, 2009, pp. 299–404. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780123743640500138

[2] D. Sánchez, L. Servadei, G. N. Kiprit, R. Wille, and W. Ecker, "A comprehensive survey on electronic design automation and graph neural networks: Theory and applications," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 28, no. 2, feb 2023. [Online]. Available: https://doi.org/10.1145/3543853

[3] D. Kaufmann, "Formal verification of multiplier circuits using computer algebra," *it - Information Technology*, vol. 64, no. 6, pp. 285–291, 2022. [Online]. Available: https://doi.org/10.1515/itit-2022-0039

[4] C. Yu, M. Ciesielski, and A. Mishchenko, "Fast algebraic rewriting based on and-inverter graphs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 9, pp. 1907–1911, 2018.

[5] A. Sayed-Ahmed, D. Große, U. Kühne, M. Soeken, and R. Drechsler, "Formal verification of integer multipliers by combining gröbner basis with logic reduction," in *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2016, pp. 1048–1053.

[6] A. Mishchenko, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, "ABC: A system for sequential synthesis and verification." [Online]. Available: http://www.eecs.berkeley.edu/~alanmi/abc

[7] N. Wu, Y. Li, C. Hao, S. Dai, C. Yu, and Y. Xie, "Gamora: Graph learning based symbolic reasoning for large-scale boolean networks," 2023.

[8] M. Fey and J. E. Lenssen, "Fast graph representation learning with pytorch geometric," *arXiv preprint arXiv:1903.02428*, 2019.

[9] M. Naumov, L. Chien, P. Vandermersch, and U. Kapasi, "Cusparse library," *GPU Technology Conference (GTC)*, 2010.

[10] M. Shan, D. Gurevin, J. Nye, C. Ding, and O. Khan, "Mergepath-spmm: Parallel sparse matrix-matrix algorithm for graph neural network acceleration," in *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2023, pp. 145–156.

[11] Y. Wang, B. Feng, G. Li, S. Li, L. Deng, Y. Xie, and Y. Ding, "Gnnadvisor: An efficient runtime system for gnn acceleration on gpus," in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI'21)*, 2021.

[12] M. Ciesielski, C. Yu, W. Brown, D. Liu, and A. Rossi, "Verification of gate-level arithmetic circuits by function extraction," in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2015, pp. 1–6.

[13] C. Yu, W. Brown, D. Liu, A. Rossi, and M. Ciesielski, "Formal verification of arithmetic circuits by function extraction," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 12, pp. 2131–2142, 2016.

[14] D. Kaufmann, A. Biere, and M. Kauers, "Verifying large multipliers by combining sat and computer algebra," in *2019 Formal Methods in Computer Aided Design (FMCAD)*, 2019, pp. 28–36.

[15] Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Transactions on Computers*, vol. C-35, no. 8, pp. 677–691, 1986.

[16] M. Ciesielski, P. Kalla, and S. Askar, "Taylor expansion diagrams: A canonical representation for verification of data flow designs," *IEEE Transactions on Computers*, vol. 55, no. 9, pp. 1188–1201, 2006.

[17] Y.-A. C. Randal E. Bryant, "Verification of arithmetic circuits with binary moment diagrams," in *32nd Design Automation Conference*, 1995, pp. 535–541.

[18] T. Raudvere, A. Singh, I. Sander, and A. Jantsch, "System level verification of digital signal processing applications based on the polynomial abstraction technique," in *ICCAD-2005. IEEE/ACM International Conference on Computer-Aided Design, 2005.*, 2005, pp. 285–290.

[19] A. Mahzoon, D. Große, C. Scholl, A. Konrad, and R. Drechsler, "Formal verification of modular multipliers using symbolic computer algebra and boolean satisfiability," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, ser. DAC '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1183–1188. [Online]. Available: https://doi.org/10.1145/3489517.3530605

[20] M. Ciesielski, T. Su, A. Yasin, and C. Yu, "Understanding algebraic rewriting for arithmetic circuit verification: A bit-flow model," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 6, pp. 1346–1357, 2020.

[21] C. Moore, N. Hanley, J. McAllister, M. O'Neill, E. O'Sullivan, and X. Cao, "Targeting fpga dsp slices for a large integer multiplier for integer based fhe," in *Financial Cryptography and Data Security*, A. A. Adams, M. Brenner, and M. Smith, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 226–237.

[22] P. Subramanyan, N. Tsiskaridze, W. Li, A. Gascón, W. Y. Tan, A. Tiwari, N. Shankar, S. A. Seshia, and S. Malik, "Reverse engineering digital circuits using structural and functional analyses," *IEEE Transactions on Emerging Topics in Computing*, vol. 2, no. 1, pp. 63–80, 2014.

[23] D. H. Bailey, "High-precision floating-point arithmetic in scientific computation," *Computing in science & engineering*, vol. 4, no. 2, pp. 54–63, 2002.

[24] J. D. Hull, S.-H. Du, T. J. Sejnowski, and P. M. Homan, "High precision financial analytics on gpus," in *Proceedings of the First Workshop on High Performance Computational Finance*, 2010, pp. 1–8.

[25] C. Paar and J. Pelzl, *Understanding Cryptography: A Textbook for Students and Practitioners*. Springer, 2010.

[26] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[27] D. Selsam, M. Lamm, B. Bünz, P. Liang, L. de Moura, and D. L. Dill, "Learning a SAT solver from single-bit supervision," *CoRR*, vol. abs/1802.03685, 2018. [Online]. Available: http://arxiv.org/abs/1802.03685

[28] Y. Ma, H. Ren, B. Khailany, H. Sikka, L. Luo, K. Natarajan, and B. Yu, "High performance graph convolutionai networks with applications in testability analysis," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*, 2019, pp. 1–6.

[29] 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.

[30] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive representation learning on large graphs," 2017. [Online]. Available: https://arxiv.org/abs/1706.02216

[31] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998.

[32] W. Sun and Z. Ma, "Count sort for gpu computing," in *2009 15th International Conference on Parallel and Distributed Systems*. IEEE, 2009, pp. 919–924.

[33] D. Merrill and A. Grimshaw, "High performance and scalable radix sorting: A case study of implementing dynamic parallelism for gpu computing," *Parallel Processing Letters*, vol. 21, no. 02, pp. 245–272, 2011.

[34] X. Xu, N. Shah, A. Evans, S. Sinha, B. Cline, and G. Yeric, "Standard cell library design and optimization methodology for asap7 pdk," 2018.