

Article

PolygonTailor: A Parallel Algorithm for Polygon Boolean Operations in IC Layout Processing

Zhirui Niu ^{1,2}, Ruian Ji ^{1,2} , Guan Wang ³, Siao Guo ³, Shijie Ye ³ and Lan Chen ^{1,2,*} 

¹ Institute of Microelectronics of the Chinese Academy of Sciences, Beijing 100029, China; niuzhirui@ime.ac.cn (Z.N.); jiruian@ime.ac.cn (R.J.)

² School of Integrated Circuits, University of Chinese Academy of Sciences, Beijing 100049, China

³ HiSilicon, Shenzhen 518129, China

* Correspondence: chenlan@ime.ac.cn

Abstract

Polygon Boolean operations are widely used in integrated circuit (IC) layout processing tasks such as design rule checking (DRC) and optical proximity correction (OPC). Single-threaded Boolean algorithms cannot meet the efficiency demand of modern IC layouts, necessitating parallel algorithms for acceleration. However, existing parallel algorithms exhibit unsatisfactory parallel speedups and limited scalability, which typically stem from an inefficient merging phase that uses generic Boolean OR operations and redundantly reprocesses all edges of polygons on grid boundaries. To solve these problems, we proposed Polygon Tailor, a novel parallel algorithm for polygon Boolean operations that employs a data-parallel strategy and a new merging approach performing incremental XOR operations solely on edges along grid boundaries, eliminating redundant computations in previous methods. This innovation drastically reduces the grid-merging time by 1–2 orders of magnitude. Compared with the parallel implementation from a commercial layout processing tool, PolygonTailor is on average $5.08\times$ faster and up to $14.36\times$ faster for OR operations that generate highly complex polygons.

Keywords: polygon Boolean operations; polygon clipping; parallel computing; computational geometry; layout processing

1. Introduction

Polygon Boolean operations, including union (OR), intersection (AND), difference (NOT), and symmetric difference (XOR), are extensively applied in the field of electronic design automation (EDA) for critical processes such as design rule checking (DRC) [1,2], mask pattern generation [3], and optical proximity correction (OPC) [4,5].

As IC manufacturing technology advances and chip integration density rises, the computational workload of polygon Boolean operations has surged. NVIDIA reports that modern chip layouts contain up to trillions of polygons [5]. Intel's research indicates that the scale of polygon data representing a microprocessor doubles every two years [6]. AMD's study [4] demonstrates a $2.3\times$ mask pattern density growth per technology node, resulting in non-linear growth of workloads, imposing more stringent requirements on algorithmic efficiency.

To handle these geometric computations, general techniques typically fall into two categories. Containing-relationship-based algorithms (e.g., the Greiner–Hormann algorithm [7]) trace polygonal boundaries based on vertex ordering and find the edges inside other polygons to determine edges to be retained. Sweep-line-based algorithms [6,8,9]



Academic Editor: Francesco Quaglia

Received: 5 January 2026

Revised: 2 February 2026

Accepted: 7 February 2026

Published: 10 February 2026

Copyright: © 2026 by the authors.

Licensee MDPI, Basel, Switzerland.

This article is an open access article distributed under the terms and conditions of the [Creative Commons Attribution \(CC BY\)](https://creativecommons.org/licenses/by/4.0/) license.

employ virtual line scanning across the plane to process edges in a strict spatial order. These serial algorithms exhibit super-linear time complexity [10], rendering them inadequate for handling the exponentially growing layout scales, making multi-threaded algorithms imperative. However, the strict ordering dependencies intrinsic to these algorithms prevent them from being embarrassingly parallel.

Some researchers have attempted to parallelize these algorithms directly. However, methods focusing on algorithm-level parallelization often face significant challenges in correctness or efficiency. For instance, ref. [11] may lead to erroneous results because of using perturbations to handle vertices on edges. Ref. [12] adopted a parallel segment tree to search intersections, which incurs excessive synchronization overhead, limiting the speedup.

Consequently, spatial partitioning strategies are adopted to realize data parallelization [13,14]. While this effectively parallelizes the Boolean operations, it introduces a critical new bottleneck: merging. Existing algorithms typically employ a generic Boolean OR operation to stitch the processed grids together, which is computationally expensive because it reprocesses polygons contacting grid boundaries and could fail to provide effective parallel acceleration when dealing with metal layers [14].

To address these challenges, this paper proposes a novel parallel algorithm. It partitions layouts into uniform grids for parallel processing and integrates results through an optimized merge algorithm. The merge algorithm utilizes an observation that only edges inside grids remain unchanged, and performs incremental computation solely on altered segments along grid boundaries, eliminating the performance bottleneck of global recomputation in other algorithms.

Evaluations on industrial layouts demonstrate over $27\times$ parallel acceleration and an average $5.08\times$ throughput advantage compared with the commercial tool in 64-thread configurations. Notably, for OR operations that generate extremely complex polygons with numerous holes—a particularly challenging scenario for existing parallel algorithms—the proposed method achieves peak speedup of $14.36\times$ over the commercial tool.

2. Related Works

Recently, the academic community has proposed multiple parallel polygon Boolean algorithms, but they all exhibit significant limitations when applied to integrated circuit layout processing:

Ashan proposed a segment tree-based parallel algorithm [12], accelerating polygon Boolean operations by parallelizing tree operations and intersection searching. However, this algorithm has two critical constraints. First, as acknowledged by the authors, excessive thread synchronization overhead limits parallel speedup to $3\text{--}4\times$. Second, the algorithm only supports one-to-one and one-to-many polygon Boolean operations, whereas layout processing typically requires many-to-many operations between two layout layers with each containing numerous polygons.

Puri developed a parallel Greiner–Hormann algorithm achieving $44\times$ parallel speedup on a 256-core cluster [11]. Nevertheless, it exhibits three fundamental flaws: Perturbations are adopted to handle vertices on edges, and perturbations could lead to non-deterministic results [9] and potential topological changes (see Figure 1), which may result in critical failures like short or open circuits when applied to layout processing. The algorithm searches all overlapping polygon pairs to implement many-to-many operations, which is computationally prohibitive for billion-polygon layouts. The algorithm requires a merging procedure for OR operation to eliminate overlaps, and our analysis reveals that the pairwise approach produces erroneous XOR/NOT results even after merging (see Figure 2).

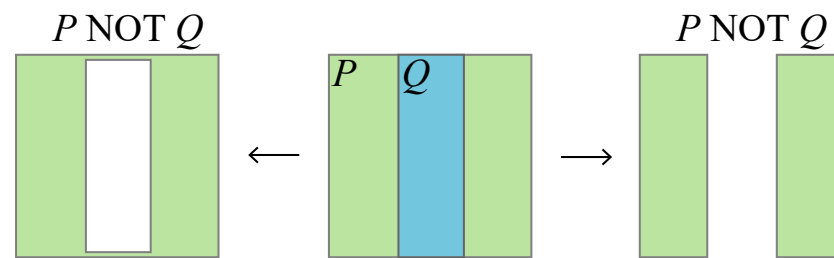


Figure 1. Different perturbations lead to different results.

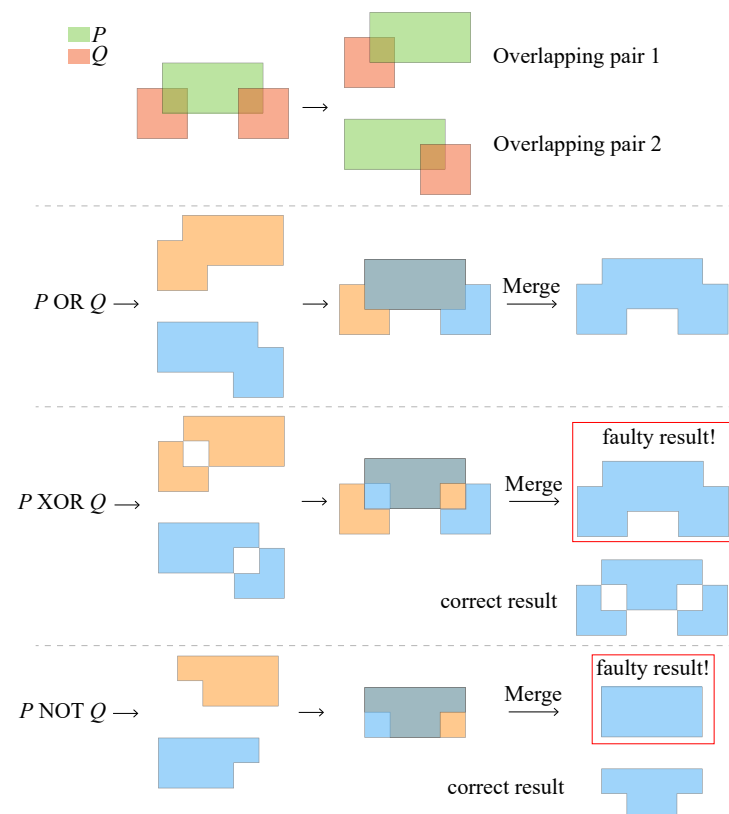


Figure 2. Different operations on polygon set P and polygon set Q . P has one polygon and Q has two polygons. If one-to-one Boolean operations are performed to each overlapping pair, the OR operation needs merging to get a correct result; XOR and NOT cannot get a correct result even if merging is conducted.

Ashan proposed a GPU-accelerated algorithm [15] which overcomes the limitations in handling vertex-edge overlaps in [11] using three filters, achieving $40\times$ speedup compared with single-threaded CPU algorithms. However, this algorithm is still limited to one-to-one Boolean operations and cannot be applied to layout processing.

Zhou proposed the Fast Candidate Edges Construction (FCEC) algorithm for IC mask Boolean operations [1], which uses a swepline method based on dual ordered arrays instead of a common binary search tree, thereby reducing the time complexity from $O(n \log n)$ to $O(n)$. Zhou also proposed a parallel implementation of FCEC: dividing the problem space into multiple subspaces by y-coordinates, computing the scanline status increments in each subspace in parallel, accumulating these increments to obtain the scanline status at the bottom of each subspace, and finally computing the final results for each subspace in parallel. This parallel algorithm has several issues: The computation of scanline status increments doubles the workload, leading to a theoretical upper bound of $p/2$ (p is the number of threads) for the parallel speedup. The paper [1] only proposes the parallel algorithm but provides no test data for the parallel implementation. The FCEC algorithm generates only

candidate edges rather than final polygons, restricting the algorithm's application scope to edge distance checks.

Puri [13] and Kullberg [14] independently developed parallel polygon Boolean algorithms based on spatial partitioning, where cross-boundary polygons are replicated in subregions they cover. Though effective for AND operations, Kullberg's analysis [14] confirms that this method inevitably produces erroneous outputs for cross-boundary polygons in NOT/XOR operations (see Figure 3).

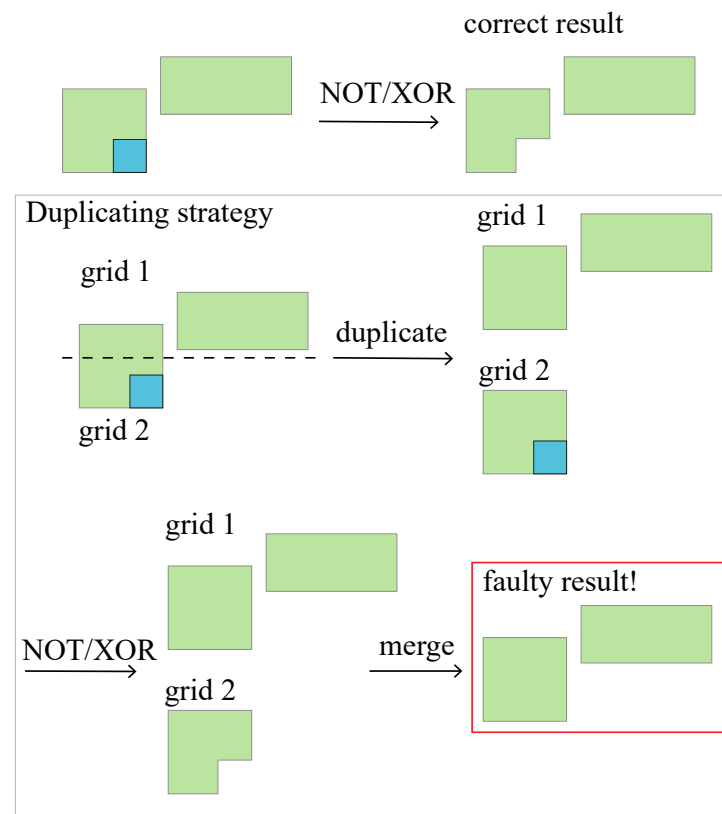


Figure 3. The duplicating strategy cannot get a correct result for NOT/XOR operations even if merging is performed.

Kullberg [14] proposed an alternative approach for processing cross-boundary polygons by clipping them against subregion boundaries, processing subregions in parallel, and merging results in subregions through generic OR operations. While this method guarantees correctness for all kinds of Boolean operations, it suffers severe performance degradation when dealing with OR operations of metal layers which generate enormous polygons with numerous holes. An example is shown in Figure 4. Intel's study indicates that certain activities of layout mask preparation require the layout being flattened and fully merged [6], and such operations involve OR operation of metal layers, generating polygons with huge spatial extents and a great multiplicity of holes. Kullberg claims that their generic-OR-based merging basically redoes the whole Boolean operation one more time when processing these immense polygons, and no speedup was yielded [14].

Commercial tools exhibit similar challenges when processing complex polygons. A commercial solution for parallel polygon Boolean operations that the authors used attains merely 5–7× speedup for OR/XOR operations under 64-thread configurations, demonstrating unsatisfactory parallelization.

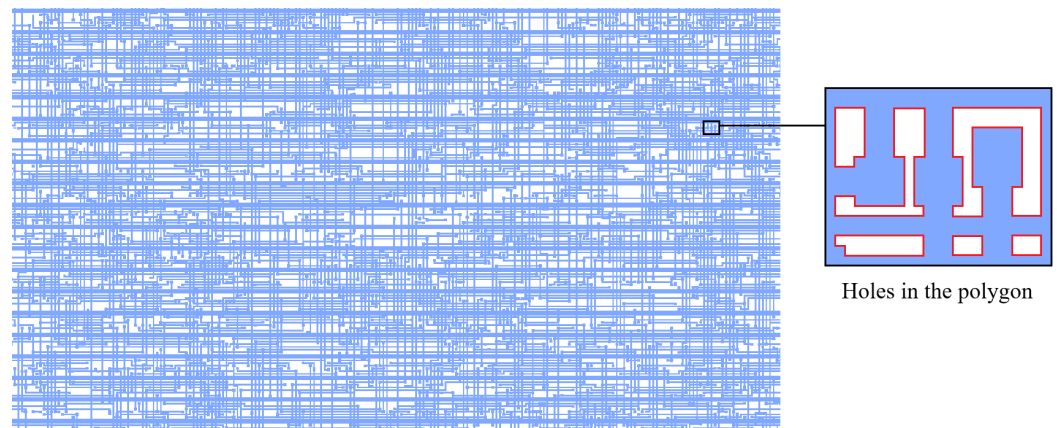


Figure 4. The OR operation of metal layers could generate a monolithic complex polygon with many holes. The boundaries of the holes are marked in red lines.

3. Algorithm

3.1. Data-Parallel Computation Framework

To achieve efficient parallel polygon Boolean operations, we propose a novel algorithm using the divide-and-conquer strategy. The layouts are partitioned into smaller grids that are more tractable to be processed concurrently on different threads. As depicted in Figure 5, the new algorithm comprises three key phases: build spatial index, clip and Boolean operations, and merge grids.

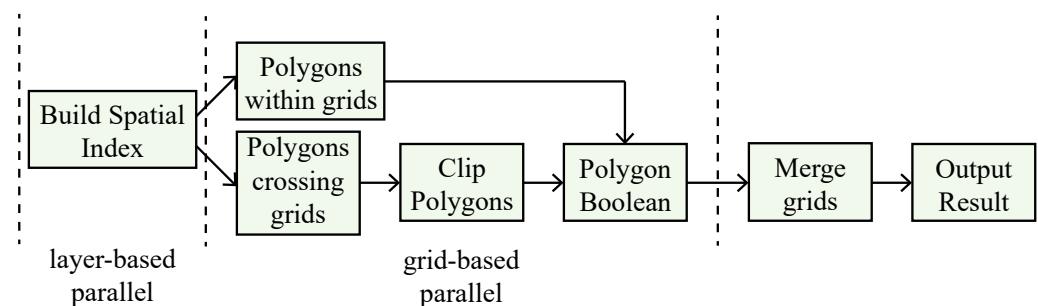


Figure 5. Flowchart of the proposed parallel polygon Boolean algorithm.

The building spatial index phase establishes the containing relationship between grids and polygons. The layout is first divided into equally sized grids. Then, the algorithm traverses all polygons and checks the spatial relationships between polygons' minimum bounding boxes and grids. Cross-grid polygons are registered in all overlapping grids, while polygons fully contained within a grid and polygon crossing grids are indexed separately in different arrays (see Figure 6). This process employs a layer-wise parallelization: different IC layout layers are processed concurrently by independent threads.

In the clipping and Boolean operation phase, polygons crossing the grids are first clipped by grid boundaries using the AND operation. Our approach fundamentally differs from Kullberg's layer-wise implementation [14] because we adopted grid-wise parallelization for clipping, enabling better thread scalability. Then, Boolean operations are performed on clipped polygons and polygons within grids. The clipping and Boolean operations utilize Boost.Polygon [6], which defines polygons as 2-d functions and treats polygon Boolean operations as the calculus of these functions. Boost.Polygon can process many-to-many Boolean operations in a single pass with $O((n + k) \log n)$ time complexity, where n is the number of edges and k is the number of intersections. It also demonstrates industrial strength for correctly handling arbitrary input polygons without preprocessing, including self-intersecting, self-overlapping, and hole-containing polygons.

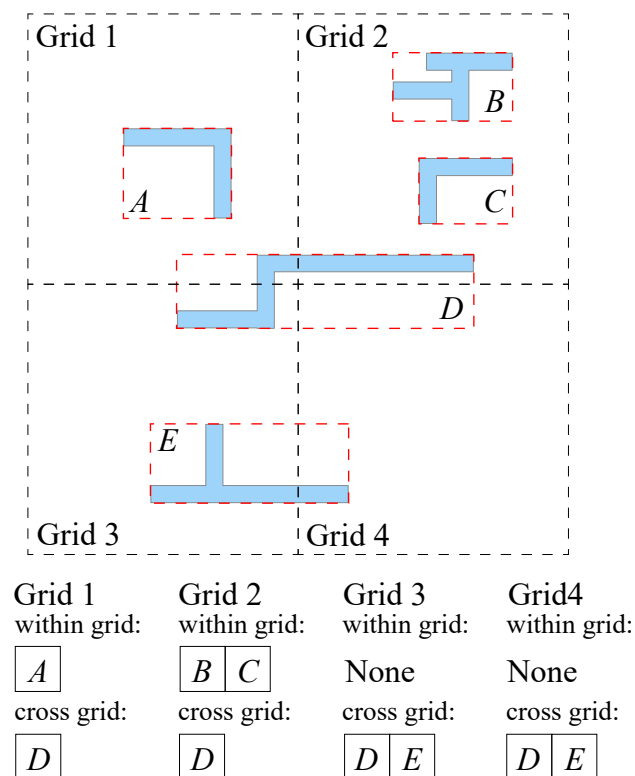


Figure 6. An example of building a spatial index for 4 grids and 5 polygons.

The grid-merging phase combines results from all grids to eliminate overlapping edges at grid boundaries and produce final outputs. A typical way to implement merging is using generic OR operations as in [14]. However, this approach incurs unacceptable runtime when cross-grid polygons are numerous, as generic Boolean OR treats all edges as new inputs, ignoring prior computations. It reprocesses all edges, even if most of them remain unchanged. According to Amdahl's Law, the parallel speedup of an algorithm is limited by its serial fraction. Our test reveals that grid merging using generic OR occupies over 50% of total runtime for OR/XOR operations and over 16% for NOT operations. To overcome this limitation, we developed a novel merge algorithm that drastically improves merging efficiency, which will be elaborated in the following section.

3.2. Novel Merge Algorithm

The inefficiency of generic-OR-based merging stems from redundant operations for non-boundary edges. After Boolean operations, polygons within grids have no overlaps, and only edges along grid boundaries require further modification. To theoretically prove that edge modifications during merging can be restricted solely to the grid boundaries without affecting the correctness of the result, we establish the following two lemmas, proven using winding number theory.

In the following text, polygons are represented by sets of vertices in counterclockwise order, while holes are represented by sets of vertices in clockwise order. All edges are directed, defined by initial vertices and terminal vertices. The merging algorithm is described only for vertical boundaries, as the process for horizontal boundaries follows the same principle.

Consider two adjacent grids: G_1 (left grid, $x \leq x_b$) and G_2 (right grid, $x \geq x_b$), sharing a common boundary L defined as $x = x_b$. Each grid contains a set of non-overlapping polygons: $P_1 = \{P_{1,1}, P_{1,2}, \dots\}$ in G_1 and $P_2 = \{P_{2,1}, P_{2,2}, \dots\}$ in G_2 . The polygons' boundaries do not cross L because they have been clipped. We define merged polygons M as the union of all polygons in P_1 and P_2 , with boundary ∂M .

Lemma 1. *Edges not on L remain unchanged after merging.*

Proof of Lemma 1. The winding number $W(p, \partial P)$ of a point p with respect to a closed path ∂P measures the number of times the path encircles p . For a polygon P with counterclockwise orientation, $W(p, \partial P) = 1$ if p is inside P , and 0 otherwise. For a hole H with clockwise orientation, $W(p, \partial H) = -1$ if p is inside the hole. For the merged region, the winding number is:

$$W(p, \partial M) = \sum_{P \in P_1 \cup P_2} W(p, \partial P).$$

Consider a point $p \in G_1$ with $x_p < x_b$, away from L . Since all boundaries of polygons in P_2 lie in $x \geq x_b$, they cannot encircle p , so $W(p, \partial P_{2,j}) = 0$ for all $P_{2,j} \in P_2$. Thus:

$$W(p, \partial M) = \sum_{P_{1,i} \in P_1} W(p, \partial P_{1,i}).$$

So, the winding number distribution inside G_1 depends only on P_1 and is not affected by P_2 . Edges in G_1 , where the winding number transitions, remain unchanged in ∂M . An analogous argument applies to edges in G_2 with $x_p > x_b$. Thus, edges not on L remain unchanged in ∂M . $\square$

Lemma 2. *Edges on L are replaced by their XOR operation results after merging.*

Proof of Lemma 2. Define S_l as the set of edges on L from P_1 , and S_r the set of edges on L from P_2 . For a point $q = (x_b, y) \in L$, we analyze its neighborhood by considering points on either side: $q_l = (x_b - \epsilon, y)$ (left, $x < x_b$) and $q_r = (x_b + \epsilon, y)$ (right, $x > x_b$), where $\epsilon > 0$ is small.

If $q \in S_l \cap S_r$, for q_l ($x < x_b$), since $q \in S_l$, q_l is inside a $P_{1,i}$, so $W(q_l, \partial P_1) = 1$. For q_r ($x > x_b$), since $q \in S_r$, q_r is inside a $P_{2,j}$, so $W(q_r, \partial P_2) = 1$. Thus, $W(q, \partial M) = 1$ in the neighborhood, indicating q is an interior point of M , not on ∂M .

If $q \in S_l \setminus S_r$, for q_l ($x < x_b$), $q \in S_l$ implies q_l is inside a $P_{1,i}$, so $W(q_l, \partial P_1) = 1$. For q_r ($x > x_b$), $q \notin S_r$, and no P_2 polygon cover q_r , $W(q_r, \partial P_2) = 0$. The left side is inside M , and the right side is outside, so $q \in \partial M$.

If $q \in S_r \setminus S_l$, for q_l ($x < x_b$), $q \notin S_l$, so $W(q_l, \partial P_1) = 0$. For q_r ($x > x_b$), $q \in S_r$ implies q_r is inside a $P_{2,j}$, so $W(q_r, \partial P_2) = 1$. The right side is inside M , and the left side is outside, so $q \in \partial M$.

Thus, $\partial M \cap L = (S_l \setminus S_r) \cup (S_r \setminus S_l)$, which is the symmetric difference (XOR) result of the edges on L from both grids. $\square$

An example depicting the two lemmas is shown in Figure 7. Based on these lemmas, we develop a novel merge algorithm that exclusively processes boundary edges through incremental computation. Compared to computationally demanding generic Boolean OR, our method replaced complex 2-d operations for all edges with tractable 1-d operations for only boundary edges. The correctness of the underlying parallel framework—comprising spatial partitioning, clipping, local Boolean operations, and merging—has been established in [14]. The proposed algorithm inherits this proven framework but replaces the computationally expensive generic OR merging with a novel boundary-based incremental approach. Lemmas 1 and 2 prove that updating edges on grid boundaries can produce the same result as generic Boolean OR. Consequently, the global correctness of the proposed algorithm is guaranteed by the correctness of the framework in [14] and the equivalence proved herein.

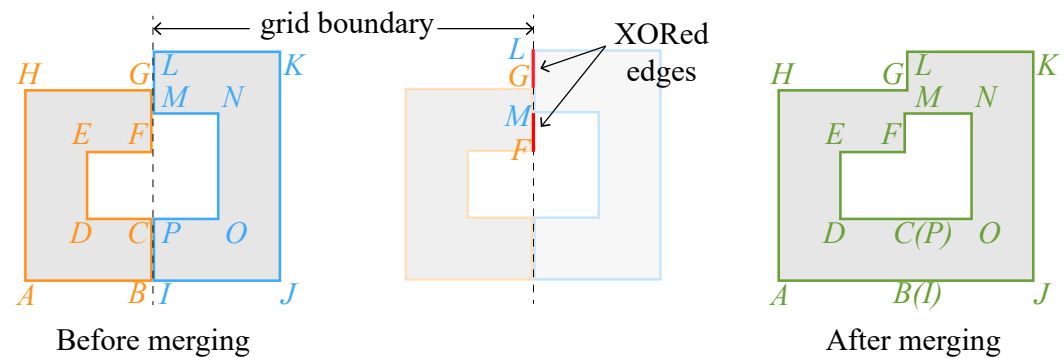


Figure 7. Two polygons have four edges, BC, FG, LM and PI , on the shared grid boundary. After merging, the edges on the boundary are LG and FM , which are the XOR result of the four edges, and non-boundary edges remain unchanged.

The algorithm traverses all edges to identify edges along the shared grid boundary. For each boundary edge, we record its initial vertex, terminal vertex, and the successive boundary vertex, defined as the initial vertex of the next boundary edge in the same polygon, as depicted in Figure 8. These successive boundary vertices indicate topological connections among boundary vertices, which are needed in subsequent procedures.

Grid boundary

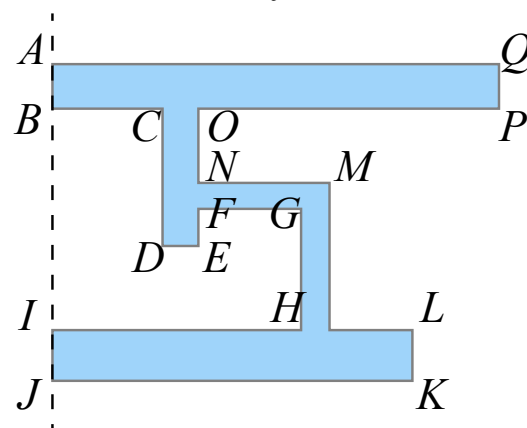


Figure 8. Vertex I is the successor boundary vertex of vertex B , and vertex A is the successor boundary vertex of vertex J .

With the boundary edges identified, the algorithm performs an XOR operation on these edges using a sweep-line algorithm adapted from [16,17]. We first sort all edge vertices by ascending y-coordinates and then sweep them from bottom to top to detect non-overlapping parts of the edges to generate edges in the XOR operation result. This XOR procedure is detailed in Algorithm 1.

After determining the remaining edges on the boundary using the XOR operation, they must be connected to non-boundary edges to form the resulting polygons. We introduce boundary vertex arrays that contain the boundary vertices of each polygon in geometric order to help construct the resulting polygons. The arrays are constructed using the following rules: an initial vertex of an edge in S_{xor} (e.g., L, F in Figure 7) connects to its terminal vertex; an initial vertex absent from S_{xor} (e.g., B, P in Figure 7) connects to its counterpart in the adjacent grid, which is next to it in the ordered *event_points* and can be easily found; and a terminal vertex (e.g., I, M in Figure 7) links to its successive boundary vertex.

Algorithm 1 Edge XOR.

```

1: Input: Boundary edges from left grid  $S_l$ , boundary edges from right grid  $S_r$ 
2: Output: Edges after XOR operation  $S_{xor}$ , binary search tree event_points
3: Sort all vertices in  $S_l$  and  $S_r$  in ascending y-coordinates and save them in event_points.
4:  $left\_count, right\_count, xor\_count \leftarrow 0$ 
5: for each vertex in event_points do
6:    $y_{now} \leftarrow$  y-coordinate of current vertex
7:   if current vertex is an initial vertex from  $S_l$  then
8:      $left\_count \leftarrow left\_count + 1$ 
9:   else if current vertex is a terminal vertex from  $S_l$  then
10:     $left\_count \leftarrow left\_count - 1$ 
11:   else if current vertex is an initial vertex from  $S_r$  then
12:     $right\_count \leftarrow right\_count + 1$ 
13:   else if current vertex is a terminal vertex from  $S_r$  then
14:     $right\_count \leftarrow right\_count - 1$ 
15:   end if
16:   if current vertex is last vertex or  $y_{now} \neq$  y-coordinate of next vertex then
17:      $new\_xor\_count \leftarrow left\_count \oplus right\_count$ 
18:     if  $new\_xor\_count \neq xor\_count$  then
19:        $xor\_count \leftarrow new\_xor\_count$ 
20:       if  $new\_xor\_count == 1$  then
21:          $current\_seg \leftarrow$  new edge in  $S_{xor}$ 
22:         if current vertex comes from  $S_l$  then
23:           Set initial vertex of current_seg as current vertex
24:         else
25:           Set terminal vertex of current_seg as current vertex
26:         end if
27:       else
28:         Set undetermined vertex of current_seg as current vertex
29:       end if
30:     end if
31:   end if
32: end for

```

Each boundary vertex array represents a polygonal boundary. If the first vertex of an array is an initial vertex from S_l or a terminal vertex from S_r , the array represents a counterclockwise boundary (a polygon); otherwise, it represents a clockwise boundary (a hole). Our algorithm provides two ways to represent polygons with holes—storing holes independently or as self-contacting polygons, offering flexibility for various EDA applications (see Figure 9). For the self-contacting polygon approach, hole arrays are embedded into the polygon array at the hole array's first vertex and the highest vertex below it in the polygon array. Since only boundary vertices are processed instead of all vertices, boundary vertex arrays are efficient for large polygons with complex holes. Algorithm 2 formalizes the construction process of boundary vertex arrays.

With boundary vertex arrays constructed via Algorithm 2, the final merged polygons are generated by traversing the boundary vertex arrays in *bva_set*. For each vertex, if its next vertex in the array is its successive boundary vertex, the algorithm retrieves their positions in the input polygon's vertex array using their pointers and copies the intervening path into the merged polygon. Otherwise, the vertex is directly included.

To illustrate the merge algorithm, consider the case in Figure 7. The algorithm begins by identifying boundary edges BC, FG from S_l and LM, PI from S_r , with terminal vertices C, G, M, I and successive boundary vertices F, B, P, L , respectively. Applying XOR operations on these boundary edges yields FM and LG . Subsequent vertex sorting produces the sequence $\{B, I, C, P, F, M, G, L\}$. Starting traversal from the first vertex B generates

the initial boundary vertex array $\{B, I, L, G\}$, counterclockwise (as B is an initial vertex from S_l), denoting an external polygon boundary. Tracing from C , the remaining lowest vertex, yields $\{C, F, M, P\}$, clockwise (as C is a terminal vertex from S_l), representing a hole. For independent storage, $\{C, F, M, P\}$ is saved separately. For self-contacting polygons, $\{C, F, M, P\}$ is embedded into $\{B, I, L, G\}$ at C and B (the highest vertex in $\{B, I, L, G\}$ lower than C), forming $\{B, C, F, M, P, B, I, L, G\}$. Final polygon generation directly extracts intermediate vertices between two boundary vertices from input polygons, such as $\{J, K\}$ between I and L in the right side polygon, producing the external boundary $\{B, I, J, K, L, G, H, A\}$ and hole $\{C, D, E, F, M, N, O, P\}$, or the self-contacting polygon $\{B, C, D, E, F, M, N, O, P, B, I, J, K, L, G, H, A\}$.

Algorithm 2 Generating boundary vertex arrays.

```

1: Input:  $S_l, S_r, S_{xor}, event\_points, flag\_independent\_hole$ 
2: Output: Set of boundary vertex arrays  $bva\_set$ 
3: Initialize  $bva\_set \leftarrow$  empty set
4: while  $event\_points$  is not empty do
5:   Initialize  $bva \leftarrow$  empty array
6:    $p\_now \leftarrow$  first vertex in  $event\_points$ 
7:   while  $bva$  is empty or  $p\_now \neq bva[0]$  do
8:     Append  $p\_now$  to  $bva$ 
9:     if  $p\_now$  is an initial vertex then
10:      if  $p\_now$  is a vertex of an edge  $S_{xor}$  then
11:         $p\_now \leftarrow$  terminal vertex of the edge
12:      else
13:         $p\_now \leftarrow$  corresponding vertex of  $p\_now$  in the adjacent grid
14:      end if
15:    else if  $p\_now$  is a terminal vertex then
16:       $p\_now \leftarrow$  successive boundary vertex of  $p\_now$ 
17:    end if
18:    Delete  $p\_now$  from  $event\_points$ 
19:  end while
20:  if  $bva$  composes a counterclockwise boundary then
21:    Append  $bva$  to  $bva\_set$  as an exterior boundary
22:  else if  $independent\_hole$  then
23:    Append  $bva$  to  $bva\_set$  as a hole
24:  else
25:    Embed  $bva$  into last array in  $bva\_set$ 
26:  end if
27: end while
  
```

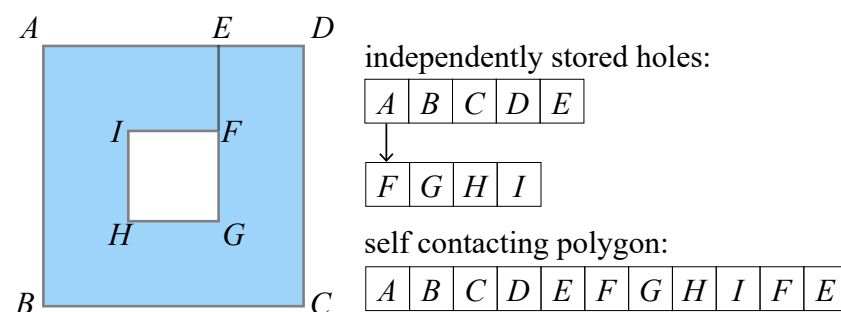


Figure 9. Two storage schemes for polygons with holes. The first approach stores the exterior boundaries and holes in separate arrays; the self-contacting polygon approach saves vertices of exterior boundaries and holes within a single array.

3.3. Complexity Analysis

The complexity of the merge algorithm for two grids is analyzed, then extended to the parallel polygon Boolean algorithm. Let n be the total polygon edges across two grids, and m the edges on the shared boundary. The key steps for pairwise merging are:

Searching boundary edges has a time complexity of $O(n)$ and a space complexity of $O(m)$. The XOR operation, dominated by sorting boundary vertices, has a time complexity of $O(m \log m)$ and a space complexity of $O(m)$. Generating boundary vertex arrays has a time complexity of $O(m \log m)$ and a space complexity of $O(m)$ since it traverses boundary vertices and operates a binary search tree. Polygon generation, traversing all vertices in polygons with edges on the grid boundary, has a time complexity of $O(n)$ and a space complexity of $O(n)$. Thus, the total time complexity is $O(n + m \log m)$, and the space complexity is $O(n)$. Since the condition $n \gg m$ generally holds in the context of integrated circuit layout processing, the time complexity is approximately linear, $O(n)$. This assumption is grounded in both geometric characteristics and empirical data. With grid sizes on the order of 100 μm , the grid dimensions significantly exceed the deep submicron technology nodes ($<100\text{ nm}$) of modern IC layouts, where a linear scale difference of approximately 10^3 times (and thus 10^6 times the area) results in a sparse grid structure. Consequently, the edges within grids significantly outnumber those along boundaries, ensuring $n \gg m$. To validate this empirically, we performed statistical analysis on the industrial layouts used in our experiments. The results indicate that with a 100 μm grid size, the number of edges on grid boundaries constitutes less than 1% of the total edge count across all layouts. This quantitative evidence ensures the algorithm's near-linear performance in practical scenarios. In contrast, the time complexity of merging with a generic Boolean OR operation is $O(n \log n)$, less efficient for large n .

Assume there are n edges, k intersections, $a \times b$ grid partitions, and p threads. Grid indexing, which traverses all polygons, yields a time complexity of $O(n)$ and a space complexity of $O(n)$. Clipping and Boolean operations have a time complexity of $O(\frac{n+k}{p} \log(\frac{n}{ab}))$ and a space complexity of $O(n + k)$.

The current merging strategy performs row-wise merging followed by column-wise merging. Although there exist other merging strategies, such as tree-based merging in [18], we chose the current approach due to its simplicity in implementation. The merging step has a space complexity of $O(n + k)$. Assuming linear time complexity for pairwise grid merging, if all polygons span at most two grids, the time complexity is $O(n + k)$. In the worst case, with a single polygon spanning all grids, the time complexity is $O((a + b)(n + k))$, derived as follows:

For a participation with a rows and b columns as shown in Figure 10, the polygons after the Boolean operation contain at most $n + 2k$ edges. Assume the edges are evenly distributed, with each grid containing $\frac{n+2k}{ab}$ edges. The initial step of row merging involves $\frac{2(n+2k)}{ab}$ edges, and the subsequent steps involve progressively $\frac{3(n+2k)}{ab}$ to $\frac{b(n+2k)}{ab}$ edges as illustrated in Figure 11. So, in total, $\frac{(2+b)(b-1)(n+2k)}{2ab}$ edges are processed, leading to $O(\frac{b(n+k)}{a})$ time complexity. Processing all rows results in $O(b(n + k))$ time complexity. For the column merging illustrated in Figure 12, each merging step processes progressively $\frac{2(n+2k)}{a}$, $\frac{3(n+2k)}{a}$ to $(n + 2k)$ edges, totally taking $\frac{(2+a)(a-1)(n+2k)}{a}$ edges, resulting in $O(a(n + k))$ time complexity. Combining the results above, the worst-case time complexity is $O((a + b)(n + k))$. Applying generic Boolean OR to merge grids yields $O((a + b)(n + k) \log(n))$ complexity, where the logarithmic term degrades performance for IC layouts with billions of polygons, as seen in Kullberg's approach [14].

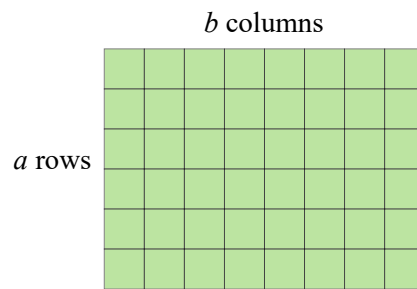


Figure 10. A uniform participation of layouts with a rows and b columns.

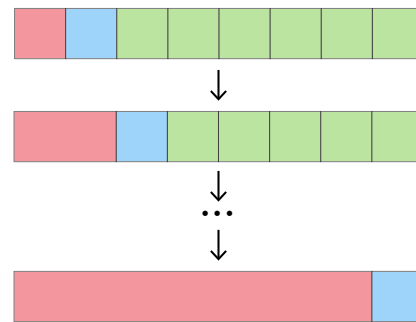


Figure 11. Row-wise merging process. Adjacent grids (red and blue) are sequentially merged from left to right. Grids awaiting merge are marked in green.

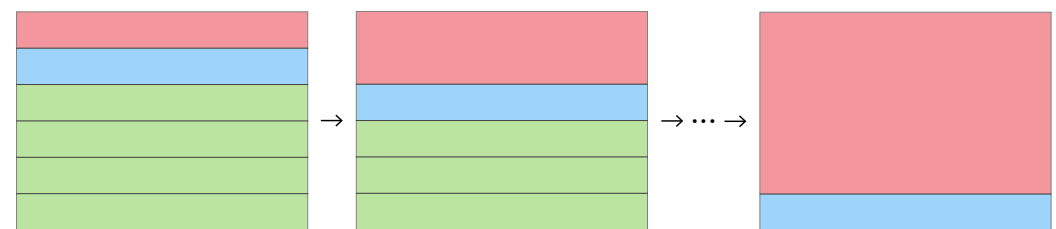


Figure 12. Steps of column-wise merging after row-wise merging. In each step, the red grid and blue grid are merged. Grids awaiting merge are marked in green.

Combining all three phases, the parallel algorithm's time complexity is typically $O(\frac{n+k}{p} \log(\frac{n}{ab}))$ and escalates to $O((n+k)(a+b+\frac{1}{p} \log(\frac{n}{ab})))$ in worst-case scenarios where polygons span all grids. The space complexity is $O(n+k)$.

4. Experiments and Analysis

Three multi-threaded Boolean operation implementations were tested: one using the novel merge algorithm for grid merging, another using generic Boolean OR for grid merging, and a solution that comes from a commercial layout processing tool. The test objects were layouts of four layers (vg, m1, etc, and aa) from real IC designs with $5 \text{ mm} \times 6 \text{ mm}$ size. The tests sequentially perform Boolean operations on the four layout layers. All tests were executed on a 2.60-GHz CPU with 64 logical processors and 256 GB of RAM. The operating system was CentOS 7. The proposed algorithm and the generic-OR method were implemented in C++ and compiled using GCC 10.4.0 with the -O3 optimization flag. The commercial tool runs on the same hardware environment. All reported runtimes are the average of 10 independent runs.

The results in Table 1 demonstrate that the novel merge algorithm significantly outperforms generic OR merging in OR and XOR operations, with modest gains in AND and NOT operations. For OR, runtime drops from 1049.65 s to 54.28 s under 64 threads, with a speedup of $36.1\times$ vs. $2.6\times$. XOR improves from 222.89 s to 85.08 s, with a speedup of

33.3 $\times$ vs. 13.5 $\times$. AND and NOT show smaller gains. This discrepancy arises from the inherent characteristics of the operators: AND and NOT operations produce fewer polygons and less workload for merging. In contrast, XOR and OR generate more cross-grid polygons, and the novel merge algorithm delivers more substantial performance enhancements for these operations. The commercial tool, similar to the implementation using generic OR merge, suffers from significant degradation in parallel speedup for OR and XOR. The new algorithm runs OR operations 14.36 $\times$ faster and XOR operations 8.01 $\times$ faster than the commercial tool.

Table 1. Runtime and Parallel Speedup of three implementations. Speedups are calculated by dividing 1 Thread runtime by 64 Thread runtimes.

Operation	New Algorithm			Generic OR Merge			Commercial Tool		
	64 Threads	1 Thread	Speedup	64 Threads	1 Thread	Speedup	64 Threads	1 Thread	Speedup
AND	23.25 s	649.60 s	27.9 $\times$	23.47 s	651.34 s	27.8 $\times$	53.74 s	1146.31 s	21.3 $\times$
OR	54.28 s	1957.08 s	36.1 $\times$	1049.65 s	2744.26 s	2.6 $\times$	779.37 s	4166.66 s	5.3 $\times$
NOT	23.94 s	660.25 s	27.6 $\times$	30.92 s	667.18 s	21.6 $\times$	59.95 s	1171.65 s	19.5 $\times$
XOR	85.08 s	2829.63 s	33.3 $\times$	222.89 s	3006.35 s	13.5 $\times$	681.61 s	5247.41 s	7.7 $\times$

To validate the optimization effectiveness of the novel merge algorithm and further analyze performance bottlenecks, we conducted runtime breakdown tests, with the results illustrated in Figure 13. We use high-resolution timers to record the precise execution time of each phase: spatial indexing construction, clipping and Boolean operations, and grid merging. For AND, merging was negligible for both implementations, due to minimal cross-grid polygons. For OR, generic OR merging dominated at 95.2% (999.03 s), while the novel merge occupies only 5.8% of run time (3.14 s), a 318 $\times$ reduction. For NOT, generic OR merging consumed 16.6% of runtime (5.12 s), while the novel merge algorithm reduced this to 0.3% (0.064 s), an 80 $\times$ improvement. For XOR, the novel merging occupies 4.5% of runtime (3.80 s), a 36 $\times$ improvement compared with generic OR merging, which takes 137.27 s. The novel merge algorithm effectively reduces merging time and improves parallel acceleration.

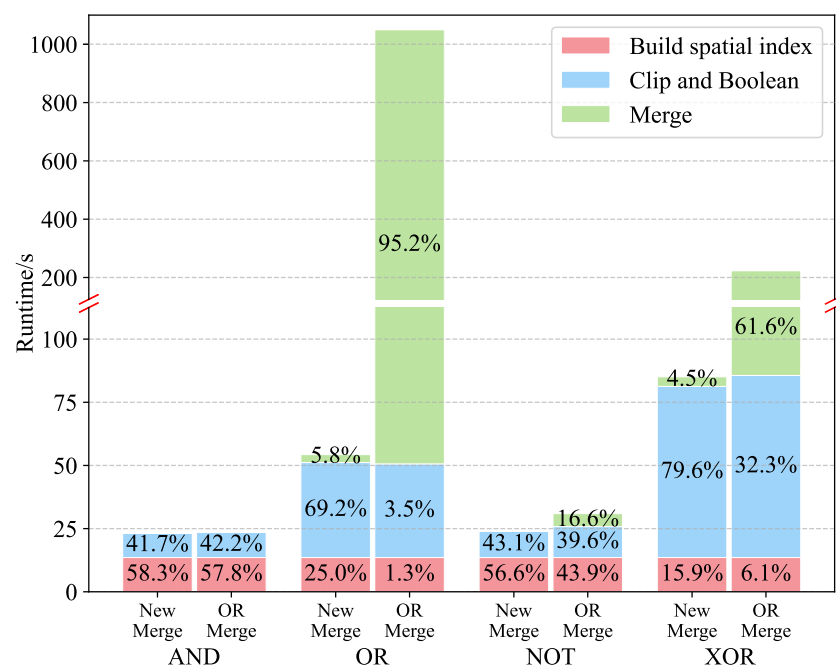


Figure 13. Runtime breakdown analysis for implementations using the new merge algorithm and generic OR merging.

The Boolean operation step using Boost.Polygon is output-sensitive. For AND and NOT operations, fewer output polygons result in shorter runtime for both Boolean operations and grid merging. Consequently, the spatial indexing construction phase with lower parallelization emerges as the primary bottleneck, accounting for over 50% of the runtime in these operations. Though Boolean runtimes are higher for OR and XOR operations, spatial indexing still represents significant time shares of 25% and 15.9%, respectively.

This limitation can be addressed by improving the parallelization of the spatial indexing phase. A possible approach is to distribute an equal number of polygons to each thread and concurrently determine the spatial relationships between polygons and grids. Such a strategy leverages the inherent independence of polygon-to-grid assignments to enable embarrassingly parallel spatial index construction and has the potential to improve the algorithm's efficiency further.

The grid size influences algorithm performance. Larger grid sizes may lead to substantial thread idling if a specific grid's runtime obviously exceeds others, forcing all other threads to wait. Conversely, smaller grid sizes reduce the likelihood of thread idling, since even the most computationally intensive grid's duration remains negligible relative to the total computation time. The clipping and Boolean step, with time complexity of $O(\frac{n+k}{p} \log(\frac{n}{ab}))$, benefits from smaller grids (larger a, b). However, excessively small grids introduce trade-offs: increased spatial indexing construction time due to higher grid counts, and elevated merging time, which has a worst time complexity of $O((a+b)(n+k))$ when dealing with complex polygons like the result of OR operations.

To determine optimal grid scale, we tested runtime across varying grid sizes, shown in Figure 14. Experimental results confirm that smaller grids enhance the speed of parallel clipping and Boolean operations, aligning with theoretical analysis. However, when the grid size falls below 100 μm , the spatial indexing time rises significantly, and the merging time for XOR/OR operations also increases because of increased a, b . The result demonstrates that the optimal grid size is 100 μm .

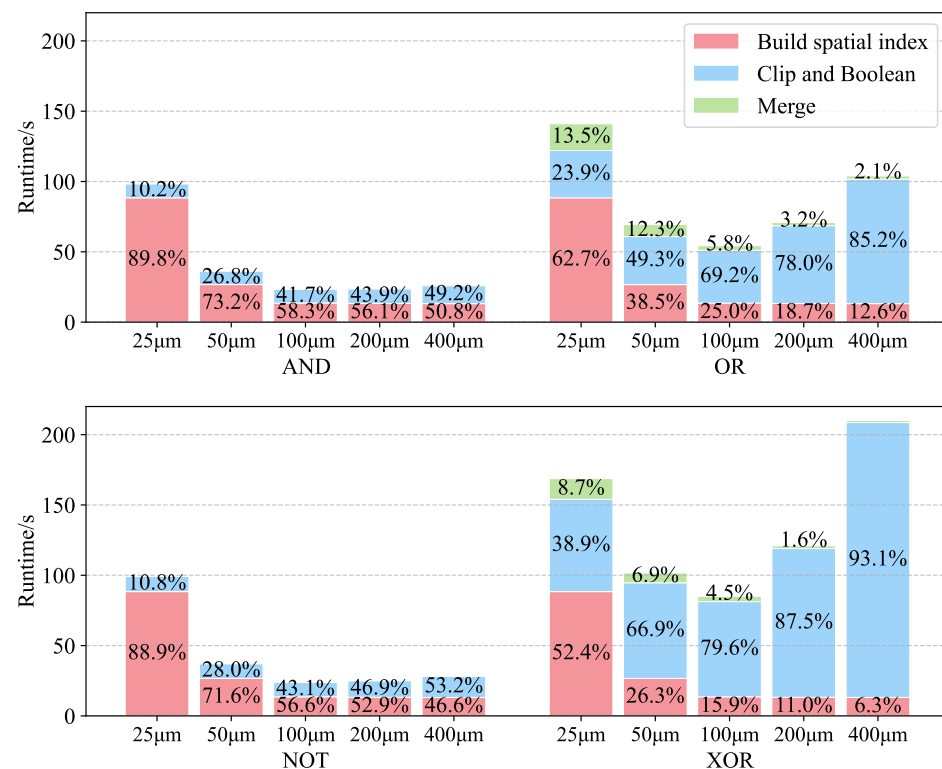


Figure 14. Runtime breakdown analysis for the novel algorithm using different grid sizes.

5. Conclusions

This paper proposes a data-parallel, multi-threaded algorithm for polygon Boolean operations. By replacing the generic Boolean OR merging implementation with an improved merging algorithm, the time for grid merging is reduced by 1–2 orders of magnitude. This advancement enables the algorithm to efficiently handle OR operations for metal layers, which all existing methods fail to do.

The algorithm has been validated on layouts from authentic IC designs. Compared to implementation using generic Boolean OR merging, it achieves over $19.34\times$ speedup for OR operations and $2.62\times$ speedup for XOR operations under 64 threads. Compared to a commercial tool, the algorithm achieves more than $2.3\times$ speedup across all operations under 64 threads, with $14.36\times$ for OR operations and $8.01\times$ for XOR operations.

Experimental results demonstrate that the novel algorithm can efficiently handle large-scale layouts and exhibits excellent parallel speedups. Currently, the spatial indexing and grid merging steps run on a single thread or few threads. Enhancing parallelism in these phases could further improve computational speed and scalability.

Author Contributions: Conceptualization, Z.N.; methodology, Z.N. and R.J.; software, S.G.; validation, Z.N., S.G. and S.Y.; formal analysis, Z.N.; investigation, Z.N. and R.J.; resources, S.Y. and G.W.; data curation, Z.N. and S.G.; writing—original draft preparation, Z.N.; writing—review and editing, L.C.; visualization, Z.N.; supervision, S.Y., G.W. and L.C.; and project administration, S.Y., G.W. and L.C. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The raw data supporting the conclusions of this article will be made available by the authors on request.

Acknowledgments: Zhirui Niu would like to express gratitude to Yukang Wang, Xiangxiang Hu and Xingdian Pan for their valuable support.

Conflicts of Interest: Authors Guan Wang, Siao Guo, and Shijie Ye were employed by the company HiSilicon. The remaining authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

References

1. Zhou, Y.; Wang, Z.; Wang, C. E2E-Check: End to End GPU-Accelerated Design Rule Checking with Novel Mask Boolean Algorithms. In Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC), Incheon, Republic of Korea, 22–25 January 2024; pp. 380–385.
2. Pais, A.P.V.; Anido, M.L.; Oliveira, C.E.T. Developing a distributed architecture for design rule checking. In Proceedings of the 44th IEEE Midwest Symposium on Circuits and Systems (MWSCAS 2001), Dayton, OH, USA, 14–17 August 2001; Volume 2, pp. 678–681.
3. Luo, T.-C.; Leong, E.; Chao, M.C.-T.; Fisher, P.A.; Chang, W.-H. Mask versus Schematic—An enhanced design-verification flow for first silicon success. In Proceedings of the 2010 IEEE International Test Conference, Austin, TX, USA, 2–4 November 2010; pp. 1–9.
4. Spence, C.; Goad, S. Computational requirements for OPC. In *Proc. SPIE 7275, Design for Manufacturability through Design-Process Integration III*; 72750U; SPIE: Bellingham, WA, USA, 2009.
5. Singh, V.K. Accelerating Computational Lithography: Enabling our Electronic Future. Available online: <https://www.nvidia.com/en-us/on-demand/session/gtcspring23-s52510/> (accessed on 4 January 2026).
6. Simonson, L.J. Industrial strength polygon clipping: A novel algorithm with applications in VLSI CAD. *Comput. Aided Des.* **2010**, *42*, 1189–1196.
7. Greiner, G.; Hormann, K. Efficient clipping of arbitrary polygons. *ACM Trans. Graph.* **1998**, *17*, 71–83.
8. Vatti, B.R. A generic solution to polygon clipping. *Commun. ACM* **1992**, *35*, 56–63.

9. Martinez, F.; Rueda, A.J.; Feito, F.R. A new algorithm for computing Boolean operations on polygons. *Comput. Geosci.* **2009**, *35*, 1177–1185.
10. Zhang, P.; Teng, X.; Fan, J.; Meng, X.; Zhao, Q.; Kang, W. Comparison of 4 vector polygon clipping algorithms in the spatial overlay analysis of GIS using simple feature model. In *Proc. SPIE 12797, Second International Conference on Geographic Information and Remote Sensing Technology (GIRST 2023)*; 127972D; SPIE: Bellingham, WA, USA, 2023.
11. Puri, S.; Prasad, S.K. A Parallel Algorithm for Clipping Polygons with Improved Bounds and a Distributed Overlay Processing System Using MPI. In *Proceedings of the 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, Shenzhen, China, 4–7 May 2015; pp. 576–585.
12. Ashan, M.K.B.; Puri, S.; Prasad, S.K. Extending Segment Tree for Polygon Clipping and Parallelizing using OpenMP and OpenACC Directives. In *Proceedings of the 53rd ACM International Conference on Parallel Processing (ICPP)*, Gotland, Sweden, 12–15 August 2024; pp. 273–283.
13. Puri, S.; Prasad, S.K. Output-Sensitive Parallel Algorithm for Polygon Clipping. In *Proceedings of the 43rd International Conference on Parallel Processing (ICPP)*, Minneapolis, MN, USA, 9–12 September 2014; pp. 241–250.
14. Kullberg, G. Parallelization of Computational Geometry Algorithms. Master's Thesis, Lund University, Lund, Sweden, 2019.
15. Ashan, M.K.B.; Puri, S.; Prasad, S.K. Efficient PRAM and Practical GPU Algorithms for Large Polygon Clipping with Degenerate Cases. In *Proceedings of the 2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, Bangalore, India, 1–4 May 2023; pp. 579–591.
16. Shamos, M.I.; Hoey, D. Geometric intersection problems. In *Proceedings of the 17th Annual Symposium on Foundations of Computer Science (FOCS 1976)*, Houston, TX, USA, 25–27 October 1976; pp. 208–215.
17. Bentley, J.L.; Ottmann, T.A. Algorithms for Reporting and Counting Geometric Intersections. *IEEE Trans. Comput.* **1979**, *C-28*, 643–647.
18. Hsu, K.-T.; Sinha, S.; Pi, Y.-C.; Chiang, C.; Ho, T.-Y. A distributed algorithm for layout geometry operations. In *Proceedings of the 48th ACM/EDAC/IEEE Design Automation Conference (DAC)*, San Diego, CA, USA, 5–9 June 2011; pp. 182–187.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.