PDRC: Package Design Rule Checking via GPU-Accelerated Geometric Intersection Algorithms for Non-Manhattan Geometry*

Jiaxi Jiang
Chinese University of Hong Kong

Lancheng Zou
Chinese University of Hong Kong

Wenqian Zhao
Chinese University of Hong Kong

Zhuolun He
Chinese University of Hong Kong

Tinghuan Chen
CUHK-Shenzhen

Bei Yu
Chinese University of Hong Kong

Abstract

With the emergence of chiplet technology, the scale of IC packaging design has been steadily increasing, making the utilization of traditional design rule checking (DRC) methods more time-consuming. In this paper, we propose PDRC, a package-level design rule checker for non-manhattan geometry with GPU acceleration. PDRC employs hierarchical interval lists within an iterative parallel sweepline framework to implement the geometric intersection algorithm, thereby finishing design rule checking tasks. Experimental results have demonstrated 30 - 50 times speedup achieved by PDRC compared with two CPU-based checkers.

1 Introduction

Advanced packaging technologies are emerging as a promising solution to address the challenges in the post-Moore's Law era. The explosive growth of advanced packages has led to I/O counts on ASICs, FPGAs, and SoC devices potentially exceeding 10,000 pins, significantly impacting the performance, capacity, and throughput of design tools [1]. One of the key challenges in packaging design tools lies in creating a precise and efficient design rule checking (DRC) workflow [2].

Design rule checking verifies a design's physical layout against the specified geometric rules, which is crucial for acceptable yield. Ensuring DRC compliance is vital for design approval, yet challenges arise in package layouts with frequently used non-Manhattan geometries. Typically, X-architecture (wire segments restricted to vertical, horizontal, and 45/135-degree diagonal directions) is employed for package/PCB routing [3, 4]. Recent advancements incorporate the any-angle routing paradigm, further reducing total wirelength and enhancing routability [5]. Nonetheless, prior design rule checking studies have not addressed these non-Manhattan design geometries [6–12].

Design rule checking is typically addressed using tools from planar computational geometry, which boasts numerous achievements [13–20]. Influenced by these successes, methodologically, researches in DRC generally fall into three categories: (1) Implementing variations of the sweepline algorithms; (2) Utilizing diverse spatial data structures; (3) Exploit various parallelism. The **Sweepline algorithm**, essential in computational geometry, includes the Bentley-Ottmann variant for geometric intersections [14] plays an indispensable role in DRC research. The Bentley-Ottmann algorithm, as slightly modified by [6, 8], improves the efficiency of design rule checking. Its axis-parallel version is employed by X-Check [11] as a sequential foundation for parallel algorithm development. OpenDRC [12] implements similar concepts to rectangle intersection report [16] for checking. **Spatial data structures** provide essential support for design rule checking via region query and neighbor search operations. Hinted quad tree [10] speeds up neighbor searching in quad-tree [13], thereby boosting design rule checking. TritonRoute-WXL [21] utilizes R-tree [17] for efficient multiple iterations of checking during the routing phase. **Various parallelism** strategies have been exploited to fully utilize the features of different modern hardware platforms. Edge-based parallelism [11], is achieved by using parallel prefix sum paradigm [20], making it well-suited for many-core GPUs. Region-based parallelism [9] is attained by partitioning the circuit into subregions for multi-core/machine devices. Other approaches, such as hierarchy-based parallelism [7] and task parallelism [8], can also accelerate performance on multi-core CPUs.

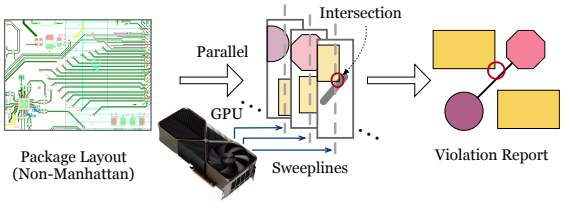

Figure 1: Illustration of PDRC: Package design rule checking.

From a computational geometry perspective, the tools researched and used by DRC fall into two principal categories: 1) tools based on partitioning the input space and data, and 2) tools utilizing spatial order defined on line segments. The **first** category includes quad-tree, R-tree and grid-like partition. Spatial data locality is attained by partitioning either the input space, as demonstrated by [9, 10], or the input data itself, as in [21]. Different sweepline variants generally fall into the **second** category. The spatial order between line segments can be determined by the coordinates of their intersection points with the sweepline [6, 8]. The efficiency of the sweepline algorithm is attained by limiting the search space to immediate neighbors. In cases where only Manhattan geometry is applied, **a notable exception combines both methods**: X-Check [11] using the Y-coordinates of horizontal segments to define the order, akin to partitioning the layout along the Y-axis. A similar approach is employed in OpenDRC's adaptive layout partition [12]. However, these exceptions are not feasible in non-Manhattan geometries due

*This work is partially supported by The Research Grants Council of Hong Kong SAR (No. CUHK14208021).

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

ACM ISBN 979-8-4007-0601-1/24/06.
https://doi.org/10.1145/3649329.3657367

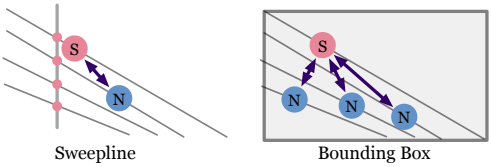

Figure 2: Examples for two geometry intersection methods.

to the failure to establish an order for unconstrained line segments. In non-Manhattan geometries, the methods in [11, 12] degrade to the first type of partitioning method, resulting in significant performance degradation. As depicted in Figure 2, with the partition-based method, three checks are needed, as the bounding box of the source segment cannot prune any other neighbor segments. In contrast, the sweepline method for detecting intersections for the source segment requires checking only with one immediate neighbor. The rationale behind this is that using bounding boxes to represent diagonal lines inevitably introduces "dead space" (empty area), thereby reducing pruning efficiency.

The sweepline algorithm can effectively solve non-Manhattan geometries, but its parallel implementation is extremely challenging [22]. A key issue is the inability to utilize all event points for the parallel prefix sum paradigm [20] (since intersection points become known only at runtime). Furthermore, the Bentley-Ottmann variant's reliance on binary trees and priority queues suggests an intrinsically sequential nature. Recently, [19] presents a new idea to remove the dependency among successive events in the sweepline algorithm with a necessary overhead required. Our PDRC develops on this concept, achieving improved work and depth by utilizing a more effective interval data structure for the GPU and extending the parallel framework. Figure 1 demonstrates the core concepts of PDRC. Our contributions are as follows:

- To the best of our knowledge, we develop the first GPU-accelerated package design rule checking engine for non-manhattan geometry;
- We present an iterative parallel sweepline paradigm for geometric intersection, enabling GPU-friendly fine-grained parallelism;
- We propose a novel data structure for intervals, named hierarchical interval lists, tailored for GPU architecture and effectively addressing stabbing queries;
- We achieve a significant speedup of design rule checking compared with several state-of-the-art design rule checkers.

2 Preliminaries

Design Rules. In this paper, to ease our algorithm design while ensuring enhanced manufacturability introduced by design rules, we propose two design rules that necessitate additional verification tools for layout validation based on previous literature [3, 4, 23]. (1) **Non-crossing and spacing rule**: Net crossings must be avoided within the same layer. Additionally, routed nets must adhere to designated spacing requirements relative to other nets, vias, and pads. Figure 3 depicts two unique types of spacing for non-Manhattan geometry: curve spacing and non-Manhattan spacing. (2) **Routing-angle and pad-entry rule**: Routing angles should be limited to either 90 degrees or 135 degrees. When routed nets enter a pad, they must avoid creating acute angles with each other. Figure 3 illustrates the routing angle rule.

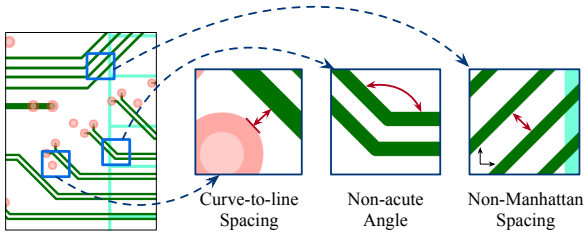

Figure 3: Non-Manhattan geometry and design rules.

Non-Manhattan Geometry. In contrast to the Manhattan geometry commonly used in Very Large-Scale Integration (VLSI) design, utilizing X interconnect architecture is favoured in packaging-level design as it reduces wirelength and vias, ultimately improving performance. The X interconnect architecture leads to the use of diagonal routing segments, and packages may also include circular vias or pads. Figure 3 illustrates the non-Manhattan geometry used in package layout.

SIMT, GPU and CUDA. SIMT (Single Instruction, Multiple Thread) is a parallel computing paradigm where each thread can operate independently with its own register state and processing elements, yet follows the same instruction stream. This approach differs from SIMD (Single Instruction, Multiple Data) by autonomously handling execution and branching for each thread, rather than manually controlling vector width. Unlike SPMD (Single Program, Multiple Data), SIMT achieves a degree of synchronized execution across threads. GPUs utilize the SIMT model for parallel computing to enhance performance. To maximize GPU performance, it's crucial to maintain similar execution paths across threads, thereby minimizing their divergence. Effective SIMT programming enables GPUs, with their thousands of cores, to outperform CPUs in terms of performance and throughput in highly parallel applications. Compute Unified Device Architecture (CUDA), developed by NVIDIA, is a groundbreaking parallel computing platform that simplifies GPU programming. It allows developers to leverage NVIDIA GPUs for a range of tasks, providing a thread hierarchy to organize parallel threads and facilitate cooperation and parallelism across various hierarchies.

3 Overview

Figure 4 provides an overview of PDRC procedures. It starts with preprocessing a non-Manhattan package layout, followed by layout expansion for spacing requirements (as explained in Section 4.1) and decomposition for algorithm needs (outlined in Section 4.2). The core of the algorithm involves establishing hierarchical interval lists for GPU (detailed in Section 4.3), crucial for addressing stabbing queries. This facilitates the parallel construction of sweepline statuses in the iterative geometric intersection sweepline framework (described in Section 4.4), where parallel intersection checks are iteratively conducted until no new intersection is detected.

4 Algorithms

4.1 Problem Formulation

We offer a brief overview of how the design rule checking tasks are translated into geometric intersection problems.

Problem 1 (Spacing check)**.** To meet specific spacing requirements, shapes are expanded so that the cumulative expansion distance between them meets or exceeds the original spacing requirement. This

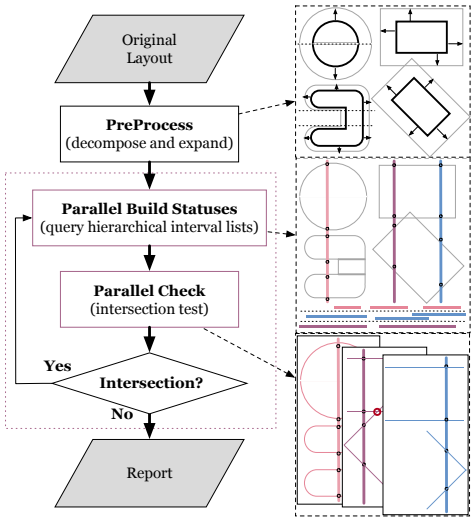

Figure 4: Overview of our algorithm procedures.

transformation makes the original task into a geometric intersection problem. For the expanded shapes' edges $\mathcal{E}^2$, we need to identify two edges e_1, e_2 with a distance $dis()$ less than or equal to zero. Formally, this can be written as:

$$\begin{aligned} &\{(e_1, e_2) \in \mathcal{E}^2\} \\ &\text{s.t. } dis(e_1, e_2) \leq 0. \end{aligned} \tag{1}$$

Problem 2 (Angle check)**.** By identifying intersection points between edges, we can calculate the angles formed at these intersections. This approach allows us to identify and examine all routing angles. For the original shape edges $\mathcal{E}^2$, we aim to determine all intersection points of edge pairs (e_1, e_2) and calculate the angles $\angle(e1, e2)$ at these intersection points to examine if they meet the requirements $\mathcal{A}$. This can be formally expressed as:

$$\begin{aligned} &\{(e_1, e_2) \in \mathcal{E}^2\} \\ &\text{s.t. } dis(e_1, e_2) \leq 0, \angle(e1, e2) \notin \mathcal{A}. \end{aligned} \tag{2}$$

4.2 Geometric Intersection with Sweepline

We introduce the foundational Bentley-Ottmann algorithm for solving the general geometric intersection problem. This algorithm efficiently addresses the problem in $O((n+k)\log n)$ time complexity, where n represents the number of geometric objects and k denotes the number of intersections [14]. Its core principle involves sweeping a line across the plane and maintaining the order of objects intersected by this sweepline. Crucially, intersections between two objects occur only when they are adjacent on the sweepline.

The Bentley-Ottmann algorithm requires geometric objects to have three properties: 1), Any vertical (horizontal) line through the objects intersects the object exactly once. 2), For objects intersecting the line, they can have a total order. 3), Given two objects, it is possible to compute the intersection point.

All convex polygons exhibit property 1, which ensures that any intersecting line with the object forms a line segment rather than a collection of discrete shapes, as depicted in the left half of Figure 5. Property 2 enables us to swiftly locate the neighbors of an object by maintaining an orderly sequence for the objects. This facilitates efficient intersection testing. Property 3 is the cornerstone of algorithms,

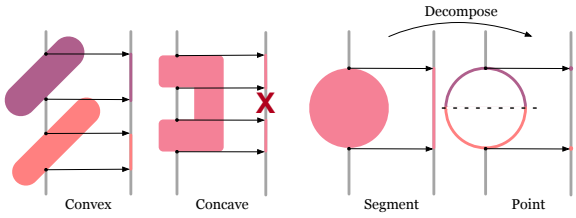

Figure 5: Bentley-Ottmann properties.

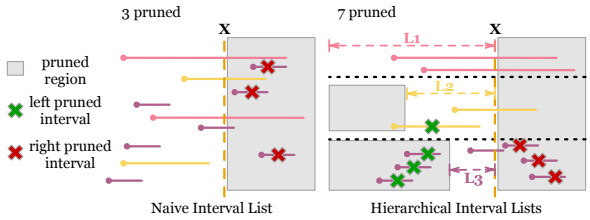

Figure 6: Comparison of stabbing query for projected segments: Without vs. with hierarchical interval lists.

providing the fundamental assurance to compute the intersection of two objects.

The three properties mentioned above essentially prefer each object's intersection with the sweepline to be a point rather than a line segment, as line segments may fail to establish a total order between them. In practical applications, objects often require modifications to adhere to the properties mentioned previously. For instance, when dealing with circles, they need to be divided into two arcs at the points where the slope is infinity. As depicted in the right half of Figure 5, a circle is divided into an upper and a lower arc, transforming the line segment intersection with the sweepline into two separate points. However, this kind of decomposition alters the original geometric intersection problem to some degree. It involves dividing the original shapes into multiple parts, consequently obscuring the inherent relational information between these shapes. A clearer understanding is that this decomposition prevents the detection of special intersection cases, namely inclusion.

To address this issue, we need to reassemble the decomposed shapes by connecting the points resulting from the division. Specifically, we reconnect the relative upper and lower intersection points to reconstruct original line segments that intersect with the sweepline. Then, we can evaluate if any overlap exists between line segments on the same sweepline to perform an inclusion check. By sorting and scanning the endpoints of the line segments, we can efficiently detect overlapping segments with a single scan in $O(n \log n)$ time complexity, where n represents the number of segments in one specific sweepline status.

4.3 Hierarchical Interval Lists for GPU

To efficiently establish sets of segments intersected by the sweepline at each position, which aids in parallel construction of sweepline statuses and enables parallel event processing, stab queries for line segments projected along the x-axis are essential. The **Build** process, comprising two substages **Label** and **Merge** as described in lines 4-9 of Algorithm 1, utilizes a novel interval data structure called hierarchical interval lists to facilitate efficiency.

The challenge with intervals, distinguished by their two endpoints, is their lack of a total order, making stabbing queries via binary search more complex. Sorting line segments by left endpoints allows pruning only when the left endpoint exceeds the query point. Additionally, consolidation of all line segments into a single data structure impedes GPU parallelization. Therefore, we assign different hierarchical labels to the line segments according to their lengths, enabling us to store segments of each hierarchy in separate arrays. Even though a total order within a single array is uncertain, the restriction on line segment lengths within the same array facilitates the use of binary search for querying, significantly limiting the return of redundant elements. Figure 6 illustrates the query results, comparing scenarios without and with hierarchical interval lists. The symbol X marks the stabbing query's location, with shaded areas and crosses showing intervals pruned via binary search. L_1, L_2, and L_3 represent the maximum lengths of the respective lists. Furthermore, by using multiple arrays of line segments that represent different hierarchies, it's possible to allocate these arrays to separate GPU threads, enabling parallel processing across hierarchies.

Compared to conventional tree-like data structures for intervals, like interval tree [15] and segment tree [16], our proposed hierarchical interval lists are better suited for GPU parallelization, enabling the acceleration of a single query through multiple threads. The distinct distribution of our stabbing queries, which is slightly different from traditional tree structure expectations, allows for advanced knowledge of query positions, thereby enhancing efficiency. Continuous ordered queries can leverage the results obtained from previous and neighbour queries (for cache reasons), thereby accelerating the querying process. Our data structure can significantly benefit from this distribution. On the other hand, accessing a tree-like structure on a GPU is less efficient for memory coalescing reasons. Contrasting with the methods used by [20] for segment queries, our algorithm simplifies the construction process, eliminating the need for complex set operations during construction. This results in a simultaneous reduction in both algorithmic and code complexity. Additionally, [20] relies on balanced trees for set union and difference, which are challenging to implement and inefficient due to irregular tree node random accesses.

4.4 Iterative Parallel Sweepline Algorithm

The sweepline-based geometric intersection algorithm does not offer a straightforward perspective to achieve parallelization. Due to the sequential nature of the Bentley-Ottmann algorithm, it processes all events in order, maintaining the order of intersecting objects with the sweepline using a binary tree, and utilizing a priority queue to maintain the order of all events. Furthermore, it is not possible to know all event positions in advance, as the algorithm generates intersection events during runtime. To achieve fine-grained parallelism which is suitable for GPU, we inevitably need to introduce additional work or memory in order to reduce the depth of the algorithm.

Building on the innovative event-parallel concept from [19], we have further developed their approach to attain position-level parallelism, specifically tailored to common data distributions in PCB and Package domains. We adopt a horizontal version of the Bentley-Ottmann algorithm for description. Rather than processing event points in ascending x-order during a left-right horizontal sweep, we initially sort and remove duplicates from all pre-known event points

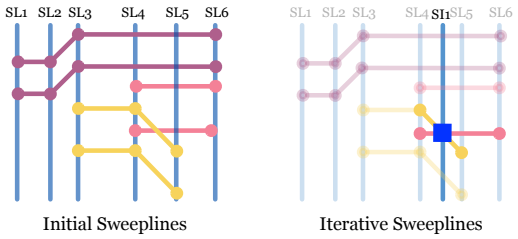

Figure 7: Iterative parallel sweepline algorithm.

(derived from segment endpoints) according to their x coordinates. This process creates initial sweeplines, illustrated as vertical blue lines in the left half of Figure 7. We then address events at each position, taking into consideration that each position may contain multiple events. The rationale behind this is related to the highly organized ball grid array and bump pad commonly used in PCB and IC packaging designs [3, 4, 23]. In the processing of each position, multiple new intersection events may emerge. The final phase of processing intersection events is iterative, new sweeplines can be derived from intersection points, as illustrated in the right half of Figure 7.

Algorithm 1 Iterative Parallel Sweepline Algorithm

Input: A set $\mathcal{O}$ of geometry objects satisfy Bentley-Ottmann three properties
Output: Geometry objects pairs which have intersections

```
Sort objects endpoints sequence 𝒫 by ascending x-coordinates;
Remove duplicated x-coordinate occurrences in 𝒫;
Initialize Hierarchical Interval Lists ℒ;
For each position in 𝒫 do in parallel                    ▷ Build
    Identify the left and right boundaries in ℒ;         ▷ Label
    Merge the segment sequence as 𝒮 from ℒ;               ▷ Merge
Endfor
For each segment sequence in 𝒮 do in parallel            ▷ Sort
    Sort segments based on the y-coordinates of the intersections;
Endfor
For each position in 𝒫 do in parallel                    ▷ Check
    Scan segments and update intersection events ℐ;
Endfor
while ℐ have new elements do                             ▷ Iterative Check
    For each new element in ℐ do in parallel
        choose merged segments and resort them;
        Handle intersection events and update ℐ;
    Endfor
end while
```

Algorithm 1 describes the details of our position-level parallelism. After acquiring the unique x coordinates of events, we utilize hierarchical interval lists to accelerate stabbing queries for intervals. During the query process, each thread uses binary search to determine the left and right boundaries within its allocated array. Following this, threads merge elements found between these boundaries within their respective arrays. Next, we sort the query results along the y-axis, enabling rapid identification of intervals adjacent to the event interval and facilitating inclusion checks. We briefly describe how to scan segments for inclusion and intersection checks. For the inclusion check, it suffices to ensure that reconnected segments

on the same sweepline do not intersect. While scanning, a set of active segments is maintained. When new segments are scanned, they might overlap with those already in the set, causing overlaps in the initial geometry. Intersection checks require handling start, end, and intersection events. In a start event, determine whether the newly added object intersects with its immediate upper and lower neighbors. For an end event, verify if the objects neighboring the ending segment intersect. At an intersection event, assess if there are intersections between the newly neighboring objects adjacent to the intersecting segments.

4.5 Summary and Discussions

To analyze the time complexity of our algorithm, we'll use the work-depth paradigm [24]. Our core algorithm consists of five stages in total: (1) **Label**. In conducting binary searches across k interval lists containing n intervals in total, and assuming the longest list has a length of $\frac{n}{\omega_k}$, the depth is $O(\log \frac{n}{\omega_k})$. The total work involved is at most $O(k \log \frac{n}{k})$. (2) **Merge**. When merging p positions, with each position expected to contain $O(\sqrt{n})$ intervals (the same estimation as in [6]), the depth is $O(\sqrt{n})$ in the worst-case scenario (where all intervals originate from the same list). Consequently, the overall work required is $O(p\sqrt{n})$. (3) **Sort**. Parallel radix sorting can be applied to each list, resulting in a depth of $O(d \log n)$, where d represents the number of digits. The total work involved is thus $O(dp\sqrt{n})$. (4) **Check**. For each position, scanning each interval list once is necessary for inclusion and intersection checks. This results in a depth of $O(\sqrt{n})$, with the total work amounting to $O(p\sqrt{n})$. (5) **Iterative Check**. In the case where there are few intersections, the "Iterative Check" stage involves $O(1)$ iterations. Each iteration repeats the process of stage **Sort** and **Check**.

Consequently, if we disregard the constant d, the total work involved is $O(p\sqrt{n})$, and the depth remains at $O(\sqrt{n})$. The space complexity is $O(p\sqrt{n})$ since it's necessary to store lists for p positions.

5 Implementation Details

A heterogeneous CPU-GPU computing platform requires deliberate coordination, incorporating techniques such as parallel GPU and CPU processing, concurrent GPU computation across streams, and overlapping data transfer with computation.

When multiple threads are writing to the results in parallel, we use a lock-free approach(atomic operations), thus avoiding the inefficiencies that come with using lock structures. The technique of utilizing overlap data transfer and computation is employed to construct hierarchical interval lists. First, the CPU is used to build collections for each hierarchy. Then, various geometries are transferred asynchronously to the GPU and sorted. At the same time, the interval lists of the previous hierarchy are also sorted during the data transfer. While the GPU sorts and constructs the hierarchical interval lists, the CPU obtains all endpoints simultaneously. This allows for concurrent computation between the CPU and GPU. We store all of the sweeplines' X coordinates in an array while simultaneously building the hierarchical interval lists during initialization. This enables computation on the GPU using different streams in the algorithm.

We utilize the thread hierarchy concept in CUDA, namely blocks and threads. For every unique position, we allocate a block where threads within it simultaneously conduct binary searches through hierarchical interval lists. To speed up the construction of the sweepline statuses, we use the block-wide sorting capabilities of CUB [25]. Then, threads in the same block can execute a parallel neighbor check on the sweepline. Distinct blocks operate independently, checking different positions. This stratified parallelism significantly increases the algorithm's concurrency and optimizes its effectiveness.

Table 1: The statistics of our benchmarks.

Benchmark	#C	#P	#N
xc7z020_t	443	1737	428
xc7z020_b	572	1390	383
xc7z030_t	447	1936	442
xc7z030_b	653	1539	416
hs3690_t	910	3529	998
hs3690_b	656	1878	496

6 Experiments

Our algorithms are developed in C++ and CUDA, and tested on a Linux machine with an Intel Xeon Silver 4210R CPU (2.40GHz) and an NVIDIA GeForce RTX 3090 GPU. Programs are compiled using NVCC 11.6 and GNU GCC 9.4. For baseline comparisons, we use three modes in KLayout [26] and implement an R-tree based checker with Boost [27], similar to KiCad [28], since KiCad does not support design rule checking as a standalone tool. This R-tree based checker implementation also provides a reliable ground truth to verify the correctness of our algorithm.

Our benchmarks consist of several industrial PCB designs, since there are few open package design benchmarks for research. Initially designed in Allegro brd format, these designs are then exported as dxf and dsn (for routing) formats compatible with KLayout and PDRC. We developed a dsn parser to facilitate the use of dsn format. To ease implementation, we simplified the shapes in PCB layouts, such as employing segments to approximate curves, akin to the approaches used in KLayout.

The statistics of our benchmarks are listed on Table 1, where #C, #P, #N denote the numbers of components, pads and nets respectively. The suffix "_t" in the table denotes top layer, while the suffix "_b" represents bottom layer. Although PCB design and package design share some similarities, such as X-architecture, package design typically involves a much larger scale compared to PCB design. Therefore, we replicate our largest benchmark, increasing its size by factors of 4, 8, and 16, respectively. As we are simply creating duplicated layouts, we have not included the replicated benchmark in Table 1. At Table 2, in "Benchmark" column, the prefixes "4", "8", and "16" indicate their increased sizes. "#Segments" column denotes the number of segments in the relevant design.

Since spacing check and angle check are quite similar. Angle check merely needs one more check for angles at each intersection, which takes little time. Only Spacing check runtime comparisons are listed in Table 2 (in milliseconds). "flat", "deep" and "tile" columns denote to the three different modes in KLayout. Flat mode denotes the naive sequential version. In the deep mode, the operations will be executed in a hierarchical manner; in the tile mode, said operations are evaluated within tiles, whereby multi-core acceleration is supported. "R-tree" column is our own implementation, similar to the design rule checker in KiCad. Each column consists of two subcolumns: "RT"

Table 2: Runtime (ms) comparisons of design rule checking.

Benchmark	#Segments	Klayout [26] flat		Klayout [26] deep		Klayout [26] tile		R-tree [27]		PDRC
		RT	Ratio	RT	Ratio	RT	Ratio	RT	Ratio	
xc7z020_t	39368	301	43.0×	272	38.9×	146	20.9×	82	11.7×	**7**
xc7z020_b	18014	232	77.3×	173	57.7×	65	21.7×	31	10.3×	**3**
xc7z030_t	45972	273	91.0×	275	91.7×	108	36.0×	90	30.0×	**3**
xc7z030_b	19500	235	78.3×	189	63.0×	69	23.0×	217	72.3×	**3**
hs3690_t	68604	825	165.0×	705	141.0×	331	66.2×	129	25.8×	**5**
hs3690_b	35082	452	150.7×	571	190.3×	192	64.0×	64	21.3×	**3**
4hs3690_t	274416	3334	222.3×	2772	184.8×	569	37.9×	1113	74.2×	**15**
4hs3690_b	140328	1849	205.4×	2240	248.9×	358	39.8×	532	59.1×	**9**
8hs3690_t	548832	6721	268.8×	5636	225.4×	1064	42.6×	3693	147.7×	**25**
8hs3690_b	280656	3731	186.6×	4445	222.3×	612	30.6×	1715	85.8×	**20**
16hs3690_t	1097664	13470	244.9×	11274	205.0×	1996	36.3×	12821	233.1×	**55**
16hs3690_b	561312	7505	220.7×	8929	262.6×	1136	33.4×	5944	174.8×	**34**
Average			143.0×		136.8×		35.3×		51.2×	

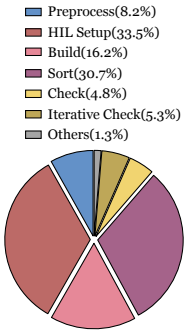

Figure 8: Average runtime breakdown.

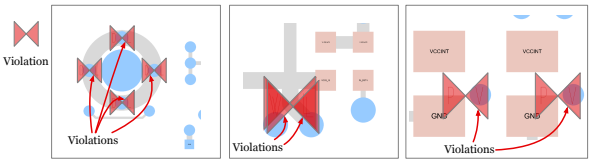

Figure 9: Illustration of real violation reports in Allegro gui. A red bow symbolizes a violation.

and "Ratio", representing runtime and ratio of runtime compared to PDRC's runtime, respectively. The final column displays the runtime of our PDRC. The "average" row is normalized against our PDRC checker, where the ratio is the geometric mean of the column, as we value all checks equally regardless of their sizes. PDRC achieves 51.2× on average speedup compared with R-tree, and 20.9 × −66.2× speedup compared with KLayout (tile mode with 8 threads).

Our algorithm demonstrates consistently enhanced acceleration in large-scale cases, due to the fine-grained parallelism of the iterative parallel sweepline algorithm. This approach results in more efficient work, $O(p\sqrt{n})$, and a shallower depth, $O(\sqrt{n})$, in contrast to brute-force parallelism, which requires $O(n^2)$ work and has an $O(n)$ depth. Figure 8 displays the runtime breakdown of CUDA kernels, highlighting that constructing hierarchical interval lists (labeled as "HIL") and sweepline statuses account for the majority of the time (since these operations involve significant amounts of memory transfer in GPU). Owing to the limited number of intersection checks needed by sweepline algorithms, **Check**'s share of the total runtime is relatively small. Since intersections are infrequent in our benchmark, the time proportion for **Iterative Check** is also small. Figure 9 depicts real violation reports in Allegro.

7 Conclusion and Roadmap

Design rule checking is a fundamental stage at the end of physical design flow. The growing complexity of package design underscores the need for fine-grained, ultra-fast parallel algorithms for design rule checking. We've implemented an iterative parallel sweepline algorithm optimized for GPUs using position-level parallelism, and efficiently done interval stabbing queries with hierarchical interval lists. In the future, we plan to collaborate with industry partners to develop more work-depth-efficient algorithms and GPU-based data structures. This will help us adapt to the demands of larger design.

References

[1] "Solving the Design and Verification Challenges of High Density Advanced Packaging," https://resources.sw.siemens.com/.
[2] C.-Y. Huang, L. Cao, K.-T. Chang, and C.-C. Wang, "High density package design platform and assembly design kit," in *Proc. MICRO*, vol. 2021, no. 1, 2021.
[3] H.-T. Wen, Y.-J. Cai, Y. Hsu, and Y.-W. Chang, "Via-based redistribution layer routing for info packages with irregular pad structures," *IEEE TCAD*, vol. 41, no. 12, pp. 5554–5567, 2022.
[4] T. Chen, S. Xiong, H. He, and B. Yu, "TRouter: Thermal-driven PCB Routing via Non-Local Crisscross Attention Networks," *IEEE TCAD*, 2023.
[5] M.-H. Chung, J.-W. Chuang, and Y.-W. Chang, "Any-Angle Routing for Redistribution Layers in 2.5D IC Packages," in *Proc. DAC*, 2023.
[6] U. Lauther, "An o (n log n) algorithm for boolean mask operations," in *Proc. DAC*, 1981.
[7] N. Hedenstierna and K. Jeppson, "A parallel hierarchical design rule checker," in *Proc. DATE*, 1992.
[8] K. MacPherson and P. Banerjee, "Parallel algorithms for vlsi layout verification," *Journal of Parallel and Distributed Computing*, vol. 36, no. 2, pp. 156–172, 1996.
[9] K.-T. Hsu, S. Sinha, Y.-C. Pi, C. Chiang, and T.-Y. Ho, "A distributed algorithm for layout geometry operations," in *Proc. DAC*, 2011.
[10] G. G. Lai, D. S. Fussell, and D. Wong, "Hinted quad trees for VLSI geometry DRC based on efficient searching for neighbors," *IEEE TCAD*, vol. 15, no. 3, pp. 317–324, 1996.
[11] Z. He, Y. Ma, and B. Yu, "X-Check: GPU-Accelerated Design Rule Checking via Parallel Sweepline Algorithms," in *Proc. ICCAD*, 2022.
[12] Z. He, Y. Zuo, J. Jiang, H. Zheng, Y. Ma, and B. Yu, "OpenDRC: An Efficient Open-Source Design Rule Checking Engine with Hierarchical GPU Acceleration," in *Proc. DAC*, 2023.
[13] R. A. Finkel and J. L. Bentley, "Quad trees a data structure for retrieval on composite keys," *Acta informatica*, vol. 4, pp. 1–9, 1974.
[14] Bentley and Ottmann, "Algorithms for reporting and counting geometric intersections," *IEEE TC*, vol. 100, no. 9, pp. 643–647, 1979.
[15] E. M. McCreight, "Efficient algorithms for enumerating intersecting intervals and rectangles," Tech. Rep., 1980.
[16] Bentley and Wood, "An optimal worst case algorithm for reporting intersections of rectangles," *IEEE TC*, vol. 100, no. 7, pp. 571–577, 1980.
[17] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *Proc. SIGMOD*, 1984.
[18] B. Chazelle and H. Edelsbrunner, "An optimal algorithm for intersecting line segments in the plane," *Journal of the ACM*, vol. 39, no. 1, pp. 1–54, 1992.
[19] A. Paudel and S. Puri, "Openacc based gpu parallelization of plane sweep algorithm for geometric intersection," in *Proc. WACCPS*, 2019.
[20] Y. Sun and G. E. Blelloch, "Parallel range, segment and rectangle queries with augmented maps," in *Proc. ALENEX*, 2019.
[21] A. B. Kahng, L. Wang, and B. Xu, "TritonRoute-WXL: The open-source router with integrated DRC engine," *IEEE TCAD*, vol. 41, no. 4, pp. 1076–1089, 2021.
[22] M. T. Goodrich, "Intersecting line segments in parallel with an output-sensitive number of processors," in *Proc. SPAA*, 1989.
[23] T.-C. Lin, D. Merrill, Y.-Y. Wu, C. Holtz, and C.-K. Cheng, "A unified printed circuit board routing algorithm with complicated constraints and differential pairs," in *Proc. ASPDAC*, 2021.
[24] J. JéJé, "An introduction to parallel algorithms," *Reading, MA: Addison-Wesley*, vol. 10, p. 133889, 1992.
[25] "CUB CUDA Libraries," https://nvlabs.github.io/cub/.
[26] "KLayout," https://klayout.de/.
[27] "Boost C++ Libraries," https://www.boost.org/.
[28] "KiCad," https://www.kicad.org/.