X-Check: GPU-Accelerated Design Rule Checking via Parallel Sweepline Algorithms

Zhuolun He
CUHK
zlhe@cse.cuhk.edu.hk

Yuzhe Ma*
HKUST(GZ)
yuzhema@ust.hk

Bei Yu
CUHK
byu@cse.cuhk.edu.hk

Abstract

Design rule checking (DRC) is essential in physical verification to ensure high yield and reliability for VLSI circuit designs. To achieve reasonable design cycle time, acceleration for computationally intensive DRC tasks has been demanded to accommodate the ever-growing complexity of modern VLSI circuits. In this paper, we propose X-Check, a GPU-accelerated design rule checker. X-Check integrates novel parallel sweepline algorithms, which are both efficient in practice and with nontrivial theoretical guarantees. Experimental results have demonstrated significant speedup achieved by X-Check compared with a multi-threaded CPU checker.

1 Introduction

Design rule checking (DRC) determines whether the physical layout of a particular chip satisfies a set of geometric design rules, which is an essential step in the physical verification flow. Typical design rules include intra-layer rules that specify a minimum width and a minimum spacing of the patterns within a layer, and inter-layer rules that define a minimum extension between shapes in different layers. A design rule checker mainly runs computational geometry algorithms [1–4] that analyze geometric relationships between primitive data objects such as polygons and edges.

Recent advancements in process technology have significantly impacted design rule checking. A practical impact is an explosion in the number of design rules that must be honored in the layout. Instead of simple width and spacing rules, modern fabrication technologies prescribe many complex contextual rules, leading to a more intensive computation workload. The continuing and growing high computational costs of DRC drive us to pursue parallel computing techniques to reduce the turn-around time for these tasks. Previous work in both academia and industry has proposed various parallel algorithms for DRC. One standard technique is to partition the layout into tiles and perform DRC on the tiles in parallel, which has been investigated since the 1980s [5]. When the layout is equipped with hierarchical information, it is also possible to exploit parallelism from the hierarchical representation, as illustrated in [6, 7]. At the edge level, Carlson *et al.* has proposed parallel algorithms for

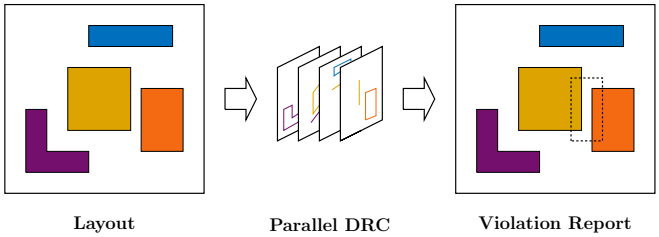


Figure 1: X-Check: GPU-accelerated design rule checking.

Manhattan geometry [8] and general (oblique) geometry [9]. The above approaches can be considered as *data parallelism*. For *task parallelism*, since design rule checking often involves more than one algorithm, Marantz [10] has developed a parallel checker that runs different checking algorithms concurrently. A systematic approach that combines both data and task parallelism is presented in [11].

Despite the fact that various parallel algorithms have been investigated, their scalability still cannot catch up with the growth of computation demand of DRC for modern designs under advanced processes. It takes more than a day and more than 2000 cores to complete an entire DRC on a 5nm design[1]. Recent years have seen GPU acceleration for various design automation stages to speed up design closure. It is pointed out that many conventional parallel algorithms do not scale beyond a few CPU cores [12], and how to better utilize the massive computing resources in GPUs needs special considerations. We detail two popular methodologies to develop efficient GPU-enabled applications. The first one is **to cast a design automation problem into another problem solvable by current tools/infrastructure**. One of the most clever ideas is DreamPlace [13], where the analytical placement problem is converted to neural network training and hence can be implemented on top of the PyTorch framework. Similarly, GATSPI [14] is a GPU-enabled gate-level simulator developed with DGL/PyTorch and customized CUDA kernels; in particular, netlists are transformed into graph objects for further operations. FastGR [15] regards the batched net routing ordering problem as a task scheduling problem, which is solved using the Taskflow [16] scheduler. By utilizing existing solvers or frameworks, developers could focus more on problem formulation and algorithm customization, without needing to build everything from scratch. The second methodology is **to design novel GPU-friendly computation kernels for some critical tasks in the design flow**. In [17], density accumulation, an essential primitive in placement, is decomposed to a density allocation phase, plus a 2D *prefix sum* phase, which is easily parallelized. GAMER [18] solves the shortest path problem in routing by iterative vertical and horizontal sweeping/relaxation, which is also conceptually a *scan* process. In [19], GPU-friendly algorithms are analyzed and implemented

*Corresponding author

ICCAD '22, October 30-November 3, 2022, San Diego, CA, USA

ACM ISBN 978-1-4503-9217-4/22/10…$15.00
https://doi.org/10.1145/3508352.3549383

[1]Statistics are provided by an industrial partner. The source is omitted due to confidentiality.

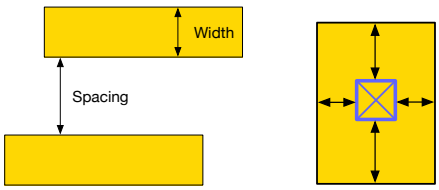


Figure 2: Typical rules: (a) *width* and *spacing* rules in a metal layer; (b) *enclosing* rule between a metal layer and a via layer.

for timing analysis, including a breadth-first search for RC delay computation, parallel levelization by advancing 'frontier', and table lookup/interpolation. We refer readers to [20] for a survey on GPU acceleration in VLSI back-end design. Moreover, GPU acceleration is also a popular topic in conference contests [21, 22].

In this paper, we propose *X-Check*, as illustrated in Figure 1, a GPU-accelerated design rule checker that parallelizes DRC via parallel sweepline algorithms. We show that many DRC tasks can be converted into a general prefix computation problem and propose a parallel sweepline paradigm to solve those problems. The proposed algorithmic paradigm is both easy to implement (GPU-friendly), and with nontrivial theoretical guarantees. We also discuss various implementation techniques to improve efficiency. In summary, our contributions are as follows:

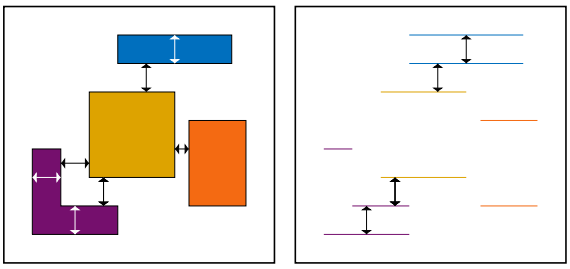
- To our knowledge, it is the first time to apply a modern general-purpose GPU to design rule checking;
- We show that many DRC tasks can be converted into general prefix computation problems, and propose a parallel sweepline paradigm to solve them;
- We implement the proposed algorithms on GPUs and fully integrate them into an end-to-end DRC flow;
- Significant speedup has been achieved on several designs of various sizes.

The rest of the paper is organized as follows: Section 2 recaps preliminaries; Section 3 introduces DRC algorithms and a parallel sweepline paradigm; Section 4 proposes two novel parallel sweepline algorithms for a series of DRC tasks; Section 5 discusses several implementation details and techniques, and Section 6 presents experimental evaluations.

2 Preliminaries

Design Rules. We illustrate a few fundamental intra-layer and inter-layer rules. Intra-layer constraints define interactions, measurements, and connectivity requirements between objects on the same layer, e.g., minimum dimensions of objects on each layer or minimum spacing between objects on the same layer, as shown in Figure 2(a). Inter-layer constraints define interactions, measurements, and connectivity requirements between objects on multiple layers, e.g., encapsulation dimensions for objects on different layers or minimum spacing between objects on different layers, as shown in Figure 2(b).

DRC Engine in KLayout. As we are going to integrate our proposed parallel DRC algorithms into the DRC flow provided by KLayout [23], we introduce the basics of the KLayout DRC engine. The DRC functionality in KLayout is controlled by a DRC script that specifies the check options and steps. KLayout organizes a layout as



(a) Distance Check (informal) (b) Distance Check

Figure 3: Distance Check. See Problems 1 and 2.

layers, which are basically collections of polygons or edges. Large layouts are first clipped into tiles to reduce memory requirements and to enable parallel processing by multiple CPU cores. In each tile, the touched objects are merged into a single object (the so-called *clean* state in KLayout). The checking tasks are then performed on the merged layers.

Parallel Computation Model. In this paper, we adopt the PRAM model with concurrent reads and exclusive writes (CREW). We use the *work-depth* (WD) paradigm [24] to analyze parallel algorithms, where *work* W is the total number of operations, and *depth* D is the length of the critical path, assuming infinitive processing resources. By the Brent's principle, the runtime T_p of an algorithm using p processors can be bounded by $T_p \leq W/p + D$.

3 Design Rule Checking Algorithms

3.1 Problem Formulation

Before diving into technical details, we first describe a general *distance check* problem that we aim to solve.

Problem 1 (Distance Check (informal))**.** A layout can be seen as a set of axis-parallel polygonal objects. The distance rule says the following: any two edges must not be closer than a predefined minimal distance. A distance violation is a pair of edges in the layout that violate the distance rule. Given a layout, the task is to report all the distance violations.

Without loss of generality, we first consider horizontal segments only. We now give a more formal definition of the above problem:

Problem 2 (Distance Check)**.** Given a set $\mathcal{H}$ of horizontal segments in $\mathbb{R}^2$, report the segment pairs from $\mathcal{H}^2$ whose horizontal projection is nonempty, and vertical distance is smaller than δ. Formally, we want to report:

$$\begin{aligned} &\{([l_1, r_1] \times y_1, [l_2, r_2] \times y_2) \in \mathcal{H}^2\} \\ &\text{s.t. } [l_1, r_1] \cap [l_2, r_2] \neq \emptyset, |y_1 - y_2| < \delta \end{aligned} \tag{1}$$

Figure 3 illustrates the our problem formulation.

3.2 Sweepline Algorithms

Technically, Problem 2 can be efficiently solved by the sweepline algorithmic framework. A sweepline algorithm can be conceptually regarded as moving a sweepline on the plane to process a set of points (a.k.a. *event points*) one by one. Suppose the event points are stored in a data structure $\mathcal{P}$ that supports a *delete-min*[2] operation in $T(\mathcal{P}_{delete-min})$ time. While $\mathcal{P}$ is not empty, the algorithm processes

[2] A *delete-min* operation finds the minimum element and removes it.

the points p in $\mathcal{P}$ by repeatedly calling the delete-min operation. For each event point p, the algorithm updates a (persistent) status data structure $\mathcal{S}$ that supports insertion, deletion, and range-report, whose time complexities are denoted as $T(\mathcal{S}_{insert})$, $T(\mathcal{S}_{delete})$, and $T(\mathcal{S}_{range-report})$, respectively. The total runtime for the sweepline algorithm can be written as

$$\begin{aligned} T_{total} &= |\mathcal{P}|T(\mathcal{P}_{delete-min}) + n \cdot T(\mathcal{S}_{insert}) \\ &\quad + n \cdot T(\mathcal{S}_{delete}) + \sum^{m} T(\mathcal{S}_{range-report}), \end{aligned} \tag{2}$$

where n is the total number of elements to be inserted/deleted to $\mathcal{S}$, and m is the total number of range-reports.

To solve Problem 2 with a sweepline algorithm, assume all the event points (i.e., endpoints of segments) are known ahead of time. We use a logarithmic time priority queue (e.g., a binary heap) to organize the event points by prioritizing their x-coordinates, which supports delete-min in $O(\log |\mathcal{P}|)$ time. For each event point, we update a self-balancing binary search tree (e.g., a BB[α] tree) that organizes horizontal segments by their y-coordinates, with $T(\mathcal{S}_{insert})$ and $T(\mathcal{S}_{delete})$ in $O(\log |\mathcal{S}|)$ time, and $T(\mathcal{S}_{range-report})$ in $O(\log |\mathcal{S}| + k)$ time where k is the number of reported elements. Specifically, for a left endpoint of a horizontal segment, we insert the segment into $\mathcal{S}$; for a right endpoint of a horizontal segment, we remove it from $\mathcal{S}$. When a segment $[l, r] \times y$ is inserted into $\mathcal{S}$, we also query $\mathcal{S}$ and report segments that are within $[y - \delta, y + \delta]$, which all violate the distance rule. Algorithm 1 summarizes the squential sweeping algorithm for distance check. Suppose there are n segments and k violations, we have

$$\begin{aligned} T_{total} &= O(n)T(\mathcal{P}_{delete-min}) + O(n) \cdot T(\mathcal{S}_{insert}) \\ &\quad + O(n) \cdot T(\mathcal{S}_{delete}) + \sum^{O(n)} T(\mathcal{S}_{range-report}) \\ &= O(n \log n + k) \end{aligned}$$

The runtime bound is optimal, as the element uniqueness problem (lower bounded by $\Omega(n \log n)$) is reducible to the problem [25], and we need $\Omega(k)$ time to report all the violations.

Algorithm 1 Sequential Sweepline Algorithm for Distance Check

Input: A set $\mathcal{H}$ of horizontal segments
Output: Segment pairs that violate the distance rule

```
Sort segment endpoints P by ascending x-coordinates
Initialize an empty BST S                ▷ use y-coordinates as keys
for all endpoint p ∈ P do
    if p is the left endpoint of a segment h = [l, r] × y then
        Range query S for [y − δ, y + δ]
        Report the corresponding segment pairs
        Insert h to S
    else
        Delete h from S
    end if
end for
```

3.3 Parallelizing Sweepline Algorithms

We present a parallel sweepline paradigm proposed in [26], the key idea of which is to regard a sweepline algorithm as computing prefix structures. We follow the notations used in [26]. Event points $p_i \in P$ are processed in a total order $\prec: P \times P \mapsto \{0, 1\}$. At each

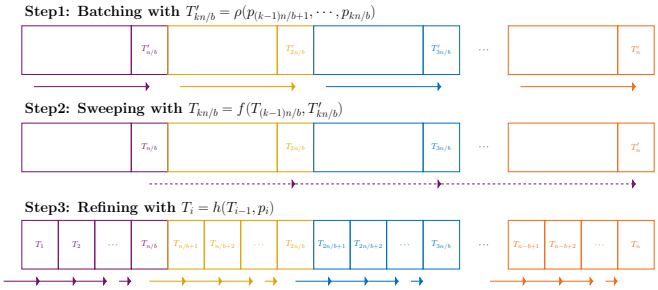


Figure 4: Parallel prefix build for sweepline algorithms in three steps: batching, sweeping, and refining. Each rectangle block represents a prefix structure, where different colors indicate different blocks. Each colored arrow represents workload of a thread. Adapted from [26].

point, our goal is to build the intermediate data structure $t_i \in T$ with the previous data structure t_{i-1} and the current point p_i using an *update function* $h : T \times P \mapsto T$ (i.e., $t_i = h(t_{i-1}, p_i)$). The initial prefix structure is t_0. In this way, we define a sweepline algorithm as a five tuple:

$$SW = \{P, \prec, T, t_0, h\}. \tag{3}$$

To describe a parallel sweepline algorithm, we further define two operators, a *fold function* $\rho : \langle P \rangle \mapsto T$ that converts a sequence of points to a prefix structure, and a *combine function* $f : T \times T \mapsto T$ that combines/reduces two prefix structures. We require f to be associative. A parallel sweepline paradigm is defined as:

$$PSW = \{P, \prec, T, t_0, h, \rho, f\}. \tag{4}$$

The essence of the parallel sweepline algorithm is to make use of the associativity of the *combine function* f. More precisely, repeatedly updating a sequence of points $\langle P \rangle$ into a sequence of prefix structures $\langle T \rangle$ using the *update function* h, is equivalent to first converting the points into (partial) prefix structures, and then combining the partial prefix structures using the *combine function* f. In [26], they propose to compute such prefix structures in three steps:

(1) **Batching**. The inputs are sorted and evenly split into b blocks. Each thread converts the consecutive n/b points in one block into a partial sum (i.e., prefix) $T'_{kn/b}$ for $k = 1, 2, \cdots, b$ using the *fold function* ρ.
(2) **Sweeping**. A single thread is invoked to sweep the b partial sums using the *combine function* f to compute the prefix structures $T_{n/b}, T_{2n/b}, \cdots, T_n$.
(3) **Refining**. The rest of the prefix structures are built using the b prefix structures built in the second step. In each block, the points are processed sequentially to update the prefix structures using h. The b blocks can be done in parallel.

Figure 4 illustrates the parallel prefix structure build. The runtime complexity of such a strategy is analyzed in [26], which depends on the complexity of the functions h, ρ, and f. We will do the analysis in Section 4 within the concrete (DRC) context.

Bootstrapping. Note that each subproblem in the refining step is of the same type as the original problem, so that we can repeatedly apply the same algorithm for each block. Such bootstrapping technique may slightly improve the runtime complexity (see Corollary 1 in [26] for details), usually by some logarithmic factor.

4 Massively Parallel Design Rule Checking

Section 3.3 describes an efficient parallel sweepline algorithmic paradigm. Now, we are going to show that design rule checking tasks fit into the prefix build framework. We claim that distance check (Problem 2) is prefix computation as described in Equation (4).

Claim 1. *Distance check is prefix computation.*

To prove the claim, we introduce two strategies to solve the problem, i.e., sweeping vertically and horizontally, in the following subsections.

4.1 The Vertical Sweeping Algorithm

Firstly, sort segments in ascending y-coordinates. We explain the algorithm by introducing the components in Equation (4).

- The event point set P includes all the y-coordinates of the segments.
- The total order $\prec$ is the total order $<$ on the y-coordinates.
- The prefix structure contains a set $\mathcal{S}$ of segments that are **below current segment within δ in y-direction**.
- The identity t_0 contains an empty set $\emptyset$.
- The *update function* h processes the segments by adding the segment to $\mathcal{S}$, and delete the segments that are below current segment by more than δ.
- For the *fold function* ρ, it suffices to first binary search for the lowest segment that is within δ to the highest segment, and then add the segments in between to the set $\mathcal{S}$.
- The *combine function* f is defined by first taking the union of the sets, and then delete the elements that are below the target segment by more than δ. Note that f is associative because the set operations are associative.

By our construction, the prefix structures contain all the candidate segments below each segment, in the sense that their distances in the y-direction are within δ. It remains to check whether each pair of segments overlap in the x-direction. Each violation will be reported by the algorithm exactly once.

We now analyze the runtime complexity of the vertical sweeping algorithm under the parallel sweepline framework. Recall that we have n events evenly split into b blocks. As an implementation trick, we store all the segments in a global array. The prefix structures only store pointers to this global array instead of explicitly storing the set elements. As a side note, the depth will grow to as large as $O(n \log n)$ if we use a persistent binary search tree for the implementation. We use s_i to denote the size of the i-th prefix structure.

(1) **Batching**. There are b blocks, and each block has $O(n/b)$ elements. The ρ function can be implemented using a binary search in the block, which takes $O(\log(n/b))$ time. The total work is $O(b \log(n/b))$.

(2) **Sweeping**. Consider the case of combining the prefix structures of the $(k-1)$-th block and the k-th block. After sweeping, the size of $T_{(k-1)n/b}$ is $s_{(k-1)n/b}$, while $T'_{kn/b}$ can have at most n/b elements. The combine function can be implemented using a binary search in these $s_{(k-1)n/b} + n/b$ elements, which takes $O(\log(s_{(k-1)n/b} + n/b))$ time. Therefore, the total work and depth are $\sum_{k=1}^{b} O(\log(s_{(k-1)n/b} + n/b))$.

(3) **Refining**. The b blocks are refined in parallel. In general, building the i-th prefix structure takes $O(\log s_{i-1})$ time. Therefore the total work is $\sum_{k=1}^{n} O(\log(s_{k-1}))$. The depth is bounded by $\max_k \sum_{i=1}^{n/b} O(\log(s_{(k-1)n/b+i-1}))$.

Note that each s_i is upper bounded by i. The prefix structures can be build in $O(n \log n)$ work and $O((b + n/b) \log n)$ depth in the worst case. When $b = \Theta(\sqrt{n})$, the depth is $O(\sqrt{n} \log n)$. This worst-case depth can be obtained by another naive solution that launches b threads to perform the n binary searches in the whole space, resulting in an $O(n \log n/b)$ depth. However, when $s_i = \Theta(\text{polylog}(i))$[3], our algorithm yields the better $O(n \log \log n/b)$ time complexity.

After building the prefix structures, each element in the prefix structures can be examined in constant time for violation check. The total work is bounded by $\sum_{i=1}^{n} s_i$. Again, the worse case complexity is $O(n^2)$, and when $s_i = o(i)$ the algorithm gives nontrivial runtime bound. Algorithm 2 summarizes the vertical sweeping algorithm.

Algorithm 2 Vertical Sweeping

Input: A set $\mathcal{H}$ of horizontal segments
Output: Segment pairs that violate the distance rule

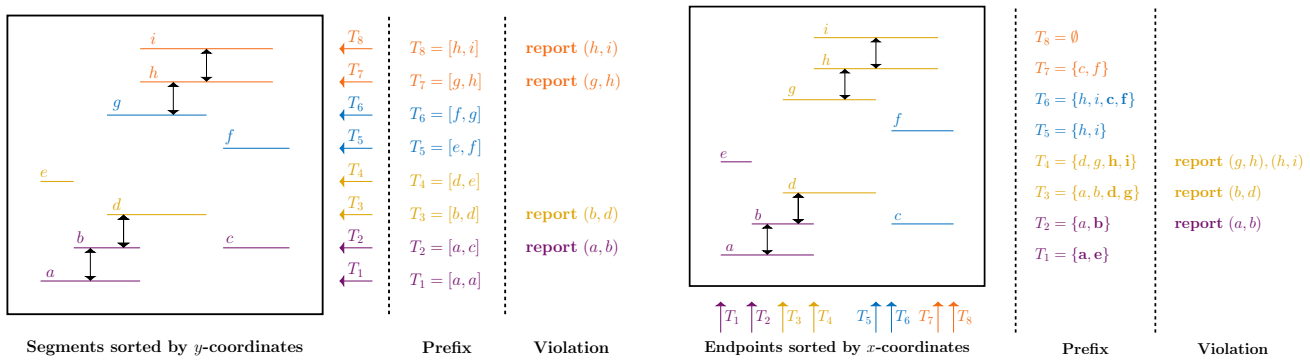
```
1:  Sort segments by ascending y-coordinates
2:  Partition the sorted segments into b blocks
3:  For each block do in parallel                          ▷ Batching
4:      Find the lowest segment that is within δ to the highest segment in the block
5:  Endfor
6:  Sweep the partial results among the b blocks           ▷ Sweep
7:  For each block do in parallel                          ▷ Refine
8:      Refine the prefix structures
9:  Endfor
10: For each prefix structure do in parallel               ▷ Report
11:     Report the violations in the prefix structure
12: Endfor
```

4.2 The Horizontal Sweeping Algorithm

For horizontal sweeping, we first sort segment endpoints in ascending x-coordinates. We also describe the components according to Equation (4).

- The event point set P contains the endpoints of the segments in $\mathcal{H}$.
- $\prec$ is the total order $<$ on the x-coordinates of the endpoints.
- The key observation here is that a sweepline algorithm maintains an 'active set' of segments. This active set of segments are those who span the current (vertical) sweepline, or equivalently, those whose left endpoints are to the left of the sweepline (i.e., have been processed), while the right endpoints are to the right of the sweepline (i.e., have not been processed yet). Therefore, we maintain two sets in the prefix structure $t = (\mathcal{L}, \mathcal{R}) \in T$: a set $\mathcal{L}$ that records the segments whose left endpoints have been processed, and a set $\mathcal{R}$ that records the segments whose right endpoints have been processed. This is natural, as the 'active set' can be easily computed by $L \setminus R$.
- The identity t_0 contains two empty sets, i.e., $t_0 = (\emptyset, \emptyset)$

[3] Some literature [2] gives $\sqrt{n}$ estimation.



(a) Vertical Sweeping **(b) Horizontal Sweeping**

Figure 5: Illustration of vertical and horizontal sweeping algorithms.

- The *update function* h processes the endpoint by adding the segment to the corresponding set. Specifically, we have

$$h((\mathcal{L}, \mathcal{R}), p) = \begin{cases} (\mathcal{L} \cup h_p, \mathcal{R}), & \text{if } p \text{ is a left endpoint} \\ (\mathcal{L}, \mathcal{R} \cup h_p), & \text{otherwise,} \end{cases}$$

  where h_p is the corresponding segment whose endpoint is p, and $(\mathcal{L}, \mathcal{R})$ is some prefix structure $t \in T$.
- The *fold function* ρ can be trivially defined as applying h for each event point p in a recursive manner. That is:

$$\begin{cases} \rho(p_1) & = h(t_0, p_1) \\ \rho(p_1, p_2, \cdots, p_n) & = h(\rho(p_1, p_2, \cdots, p_{n-1}), p_n) \end{cases}$$

- The *combine function* f is defined using set union, i.e.,

$$f((\mathcal{L}_1, \mathcal{R}_1), (\mathcal{L}_2, \mathcal{R}_2)) = (\mathcal{L}_1 \cup \mathcal{L}_2, \mathcal{R}_1 \cup \mathcal{R}_2)$$

  Note that f is associative because the set union operator $\cup$ is associative.

In this way, we also successfully relate the distance check problem to prefix computation, which can be parallelized using the strategy in Section 3.3. We now analyze the time complexity. Assume we organize sets in self-balanced binary search trees (specifically, on their y-coordinates) that support common logarithmic time operations. Merging two binary search trees of sizes m and n takes $O(m + n)$ time.

(1) **Batching**. There are b blocks, and each block has $O(n/b)$ elements. The total work is $O(b \cdot n/b \log(n/b))$, and the depth is $O(n/b \log(n/b))$.
(2) **Sweeping**. Consider the case of combining the prefix structures of the $(k-1)$-th block and the k-th block. After sweeping, $T_{(k-1)n/b}$ may contain at most $(k-1)n/b$ elements, while $T'_{kn/b}$ can have at most n/b elements. With our assumed tree operation bounds, it costs $\sum_{k=1}^{b} O(kn/b) = O(bn)$ work and the same amount of depth.
(3) **Refining**. The b blocks are refined in parallel. Consider the k-th block, where each prefix structure can have at most kn/b elements. Therefore the total work is $\sum_{k=1}^{b} O(n/b \log(kn/b)) = O(n \log n)$. The depth is $O(n/b \log n)$.

By combining the three stages, we have total work $O(n(b + \log n))$ and depth $O(n(b + \log n/b))$. When $b = \Theta(\sqrt{\log n})$, the depth is $O(n\sqrt{\log n})$. This runtime bound is worse than that of the vertical sweeping algorithm, but it maintains the y-coordinates of the segments in order. To report violations from the prefix structures, it suffices to perform two predecessor/successor searches and report violations within the range, which costs $O(\log n + k)$ work and time, where n is the size of the prefix structure, and k is the number of reported elements. Recall that such a range search takes $O(n)$ work in the vertical sweeping algorithm.

4.3 Summary and Discussions

Both algorithms proposed in previous sections give nontrivial runtime guarantees. We summarize them in the following theorem:

Theorem 1. *Assume $s_i = \Theta(\text{polylog}(i))$. Distance check can be solved in $O(n \cdot \text{polylog}(n))$ work and $O(\sqrt{n} \cdot \text{polylog}(n))$ depth, or in $O(n \log n)$ work and $O(n\sqrt{\log n})$ depth.*

We then show that many DRC tasks are distance check.

Claim 2. *Width check is distance check.*

Proof. Let $\mathcal{H}$ be the horizontal segments of a polygon. Let δ be the minimum width constraint. Then, a distance check reports all the horizontal segment pairs that violate the width constraint. Similarly, rotate the polygon by 90°. Now a distance check reports violation between (originally) vertical segments. □

Corollary 1.1. *Width check can be solved in $O(n \cdot \text{polylog}(n))$ work and $O(\sqrt{n} \cdot \text{polylog}(n))$ depth.*

With almost identical arguments, we have following corollaries.

Corollary 1.2. *Space check can be done in $O(n \cdot \text{polylog}(n))$ work and $O(\sqrt{n} \cdot \text{polylog}(n))$ depth.*

Corollary 1.3. *Enclosing check can be done in $O(n \cdot \text{polylog}(n))$ work and $O(\sqrt{n} \cdot \text{polylog}(n))$ depth.*

We also illustrate the vertical and horizontal sweeping algorithms in Figure 5. Both vertical and horizontal sweeping algorithms do not dominate each other: they achieve either better work efficiency or better depth bound. From our perspective, the vertical sweeping algorithm appears to be more promising since it provides polynomially better theoretical depth ($\tilde{O}(\sqrt{n})$ v.s. $\tilde{O}(n)$), which is the main reason why we turn to GPU acceleration. Besides, the vertical sweeping algorithm looks easier to implement, as the prefix construction

mainly relies on 1D binary search. In contrast, for horizontal sweeping we have to count on efficient set operations (set union and set difference). This is a bit surprising at the first glance, because we often use a horizontal sweeping strategy for sequential implementation (see Section 3.2). We would like to argue that the phenomenon provides an intuitive yet important insight for parallel algorithm design: suppose the problem can be decomposed along both an 'easier' direction and a 'harder' direction, *it is better to decompose a problem by the* 'simple' *direction for parallelism, and leave the* 'complex' *work to each individual processors.* In the distance check case, since the segments are horizontal, the x- and y-coordinates are not equivalent. The y-coordinate is the 'easier' direction because each segment has only one y-coordinate, which forms a total order. This also implicitly enables the use of two pointers to indicate a range (recall how do we represent a prefix structure in the vertical sweeping algorithm). On the contrary, the x-coordinate is the 'harder' one, as each segment has two endpoints with different x-coordinates, and thus there is no global total order of the segments. To deal with the complexity, we proposed to use two sets to maintain the left endpoints and right endpoints separately, but it inevitably complicates the algorithm. Note that the emphasis is different from sequential algorithm design: in the sequential sweepline algorithm, we would like to use the sweepline paradigm to look after the complex x-coordinates and leave the simple y-coordinates to the status data structure $\mathcal{S}$ (defined in Section 3.2) that we want to maintain for efficient queries.

5 GPU Implementation

The massive parallelism exposed in the vertical sweeping algorithm mainly comes from the divide-and-conquer paradigm, which is conceptually GPU-friendly. Whenever the inputs are split into blocks and processed in parallel, we launch multiple GPU kernels to perform the jobs concurrently. Nevertheless, we would like to introduce several implementation details/considerations that we find crucial to obtain satisfying performance.

5.1 Dynamic Algorithm Selection

GPU acceleration is not a free lunch. To run applications on GPUs, we inevitably have to move input data from host memory to device memory, launch GPU kernels, wait for synchronization, and move results back from device memory to the host memory. These operations have overhead. When the degree of parallelism is not high enough to make full utilization of GPU threads, the overhead might dominate the overall runtime and decelerates the whole program.

One straightforward way to compensate for such overhead is to make a dynamic decision of whether executing on GPU helps. For design rule checking, our experience is to estimate the parallelism degree by counting the average number of edges per polygon. The more edges there are in a polygon, the higher chance it has to gain performance from GPU acceleration. Accordingly, we develop a simple dynamic algorithm selection strategy that first calculates the average number of edges per polygon for each tile. If the number is higher than a threshold, we send it to the GPU branch for parallel checking; otherwise, we simply run a sequential checking (i.e., CPU branch) for the tile. The strategy is simple yet effective, as it helps X-Check to match the efficiency of CPU checkers for small/simple tasks. The detailed comparisons are shown in Section 6.3.

5.2 Sorting Strategy Selection

Sorting is an essential step in our sweeping algorithm. One of the available efficient sorting procedures comes from the *thrust* library, i.e., `thrust::sort`. In the actual program, we need to sort an array of **`struct`**s by the desired keys, which are some specific fields in the **`struct`**s. For example, when the **`struct`**s represent edges, we might want to sort them by the x-coordinates for the vertical edges, and by y-coordinates for the horizontal edges (recall how do we sweep them in the algorithm). The default way to implement is to pass a *comparison function object* as an argument to the `thrust::sort` function. Specifically, `thrust::sort` runs a *merge sort* procedure for such use cases.

Internally, *thrust* also provides a *radix sort* procedure that works for numeric data types (e.g., **`int`**) and default comparators. Therefore, an alternative solution is to copy the keys out, sort the keys using *radix sort*, and permute the **`struct`**s according to the sorted results. We call such a solution a Copy-Sort-Permute (CSP) strategy. The code snippet to implement the CSP strategy with *thrust* procedures is shown in Listing 1.

Listing 1 Copy-Sort-Permute to sort long arrays.

```
template <typename S>
void sort_long_arrays(S *array, int n) {
  int *keys;    // the buffer for keys
  int *indices; // the buffer for indices
  S *tmp;       // the buffer for permutation
  // step 0: properly allocate the buffers
  cudaMallocManaged(...)...
  // step 1: Copy
  for (int i = 0; i < n; ++i) {
    keys[i] = array[i].key;
    indices[i] = i;
  }
  // step 2: Sort
  thrust::sort_by_key(keys, keys+n, indices);
  // step 3: Permute
  thrust::copy_n(
      thrust::make_permutation_iterator(
          array, indices),
      n, tmp);
  thrust::copy_n(tmp, n, array);
}
```

Intuitively, the CSP strategy should only be used with long arrays because it definitely involves more steps and extra work, which would not be desired if sorting itself is already fast enough. In our practice, we only use CSP for arrays of size larger than 8000. Detailed comparisons and more experimental evaluations are shown in Section 6.3.

5.3 Kernel Granularity

It is possible to allocate GPU threads at various granularities; that is, each GPU thread can be responsible for solving a subproblem of various scales. After parallel prefix computation, the primary decision we face is how to report violations from the prefix structures and assign those computational tasks to GPU threads. From coarser-grained to finer-grained, we might assign GPU threads for

1) tile-wise tasks, 2) polygon-wise tasks, 3) tasks indicated by a prefix structure, and 4) a single violation examination task. To allow adequate parallelism, option 1) might not be a good choice. To balance the workload of each thread, options 2) and 3) might not be desired, as the sizes of polygons differ significantly, and the sizes of prefix structures may vary by $\Theta(n)$ in the extreme case, where n is the number of segments in the problem input. Therefore, we decided to implement option 4) in our practice, where we use the unique thread id as the *global* offset to locate its task. Specifically, for the t-th thread, its task is the q-th task in prefix T_p, such that $t = \sum_{i=0}^{p-1} |T_i| + q$.

6 Experimental Results

We implement our algorithms in C++ and CUDA, and conducted experiments on an Intel Xeon 2.90 GHz Linux machine with 128 GB RAM and one NVIDIA GeForce RTX 3090 GPU. We compile our programs with NVCC 11.4 and GNU GCC 10.3. Since KLayout [23] (version 0.26.6) is utilized to complete the end-to-end DRC flow, we use the default DRC functionality in KLayout (8 threads) as the baselines. The designs tested in the experiments are all synthesized from the OpenROAD project [27].

6.1 Runtime Comparisons

We first compare the overall runtime of design rule checks between X-Check and KLayout. The results are shown in Table 1 (*width* check) and Table 2 (*space* check, *enclosing* check).

Width check is the most simple task among the checks, as it examines violations within each polygon. Although it is less meaningful to discuss speedup when the program already runs fast, we still observe mild performance gain (1.13× and 1.18×) in the two largest cases (i.e., `Metal1` of `bp_be` and `Metal1` of `bp`). Besides, despite the dynamic algorithm selection process, such overhead is negligible (<0.1s) in all the cases.

For *enclosing* check and *space* check, the CPU version takes a much longer time to complete. Therefore, X-Check achieves a much higher speedup in these cases. For *enclosing* check, GPU-enabled X-Check achieves up to 1257.76× speedup, with an average of 61.36×. For *space* check, X-Check offers up to 280.66× speedup and an average of 45.00× improvement. The significant speedup confirms the effectiveness of our proposed parallel sweepline paradigm.

6.2 Runtime Breakdown

We care about the breakdown of runtime because 1) we want to understand where does the speedup come from, and 2) want to foresee where is the new bottleneck for potential further speedup.

Width Check Runtime Breakdown. As we have achieved mild speedup for width check, it is desired to profile the application after GPU acceleration for further analysis. Therefore, we collected the runtime statistics of each thread for the largest test case `bp`, with a particular interest in the comparison between the *merge* stage and the *check* stage. The results are shown in Figure 7. Each horizontal bar is for one thread, where the purple portion is for the *merge* stage, and the gold portion is for the *check* stage. For KLayout, the *check* stage takes from 21.3% to 56.6% of the runtime, with an average of 39.8%. After GPU acceleration, the *check* stage in X-Check takes from 1.5% to 21.4% of the runtime, with an average of 6.5%. The result matches that in Table 1, indicating the source of the current speedup, as well as explaining the limited performance gain.

(a) KLayout (b) X-Check

Figure 7: Runtime breakdown of *width check* on `Metal 1` of the `bp` design. The purple and gold portions are for the *merge* and the check stages, respectively.

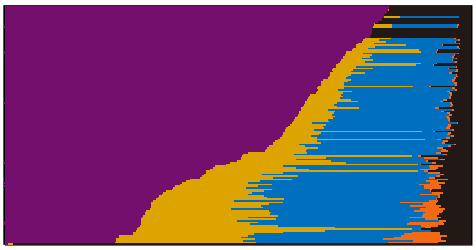


Figure 8: Runtime breakdown of *enclosing check* on `Metal 1` of the `bp` design. The purple portion is for *merge*, gold for *sort*, blue for *prefix build*, orange for *violation report*, and black for the rest, respectively.

Enclosing Check Runtime Breakdown. We are also curious about the runtime breakdown of the slow cases. Therefore, we also profiled X-Check on the enclosing check for the `bp` design. The results are shown in Figure 8. In the figure, each horizontal bar is for one tile, where the purple portion is for *merge*, gold for *sort*, blue for *prefix build*, orange for *violation report*, and **black** for **the rest**, respectively. Some tiles do not have valid enclosing checks to be performed, so there is no time spent on *sort*, *prefix build*, and *check*. The lowest bar has a substantial portion for 'the rest' because it is the first tile and carries some warm-up jobs for GPU. From the figure, the *merge* stage still takes a significant portion of time (up to 82.5% and averaged 55.9%), indicating the new runtime bottleneck after GPU acceleration of the sweepline algorithm for violation report.

6.3 Ablation Study

In this section, we further investigate the effectiveness of some implementation techniques we discussed in Section 5.

Dynamic Algorithm Selection. As introduced, it is not desired to invoke GPU execution if the estimated parallelism is limited. To illustrate the importance of such a strategy, we compare the width check runtime for `Metal 2` of all the designs. For these cases, the average edges per polygon are small - it is unlikely to have performance gain by involving GPU computation. The experimental results, as shown in Figure 9, have confirmed the case.

Sorting Strategy Selection. Sorting strategies also affect the runtime performance. To demonstrate, we compare the runtime of enclosing check using merely `thrust::sort` (i.e., merge sort), merely Copy-Sort-Permute strategy, and a mixed strategy that switches to CSP when the array size is larger than a predefined threshold (8k in our practice). As shown in Figure 10, the mixed strategy indeed outperforms both single strategies.

Besides, we further tested sorting synthetic arrays. In this setting, the array contains `structs` whose sizes are 48 bytes. The array lengths vary from 2 to at most 2^{25}, and we sort them using both

Table 1: Runtime Comparisons of *Width* Check

Design	Layer	#Tiles	#Polygons	#Edges	#Edge/Polygon	Width Check Time (s) KLayout	X-Check	Speedup
gcd	Metal1	1	391	24440	62.5	<0.1	0.1	-
	Metal2	1	1229	4916	4.0	<0.1	<0.1	-
aes	Metal1	16	17739	2059906	116.1	2.9	3.0	0.97×
	Metal2	16	76007	304028	4.0	0.2	0.1	-
bp_be	Metal1	56	34747	27245522	784.1	21.9	19.3	1.13×
	Metal2	56	393834	1575336	4.0	0.4	0.4	-
bp	Metal1	144	107706	52595418	488.3	38.9	33.0	1.18×
	Metal2	144	833588	3334352	4.0	0.9	0.9	-
Average								1.09×

Table 2: Runtime Comparisons of *Enclosing* Check and *Space* Check

Design	Layer	Enclosing Check KLayout	X-Check	Speedup	Space Check KLayout	X-Check	Speedup
gcd	Metal1	38.4	2.4	16.00×	12.6	2.4	5.25×
	Metal2	2.5	2.5	1.00×	6.4	2.4	2.67×
aes	Metal1	15 470.4	12.3	1257.76×	4493.8	67.5	66.57×
	Metal2	2227.0	14.5	153.59×	2778.5	9.9	280.66×
bp_be	Metal1	66 194.6	128.6	514.73×	6718.7	123.7	54.31×
	Metal2	3089.2	147.4	20.96×	4171.5	16.6	251.30×
bp	Metal1	98 370.4	235.3	418.06×	14 019.7	233.4	60.07×
	Metal2	3958.7	276.6	14.41×	5164.4	65.9	78.37×
Average				61.36×			45.00×

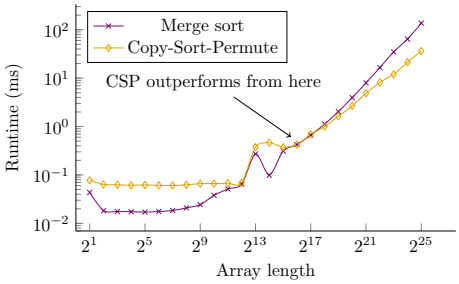


Figure 6: Runtime comparisons of merge sort and the CSP sorting strategy. CSP outperforms merge sort when the input arrays are large.

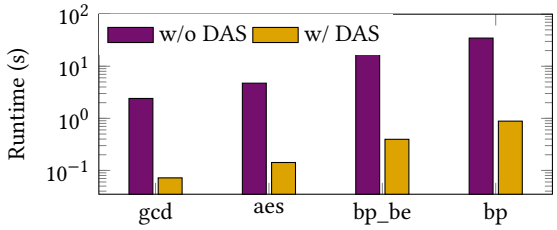


Figure 9: Runtime comparisons of width check on `Metal 2`. Runtimes are in log scale. For the sparse tiles, dynamic algorithm selection significantly reduces runtime.

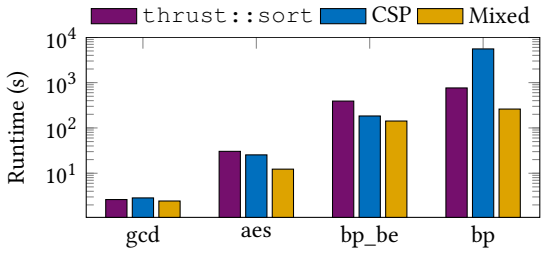


Figure 10: Runtime comparisons of enclosing check on `Metal 1` using different sorting strategies. Runtimes are in log scale. The mixed strategy achieves the fastest runtime in all cases.

merge sort and the copy-sort-permute (CSP) strategy and compare the performance. The results are shown in Figure 6, and note that both axes are in log scale. As can be seen, the CSP strategy runs significantly faster than merge sort for large arrays. CSP outperforms merge sort when the input arrays are large enough. However, CSP runs slower for smaller arrays as the overhead cannot be ignored for those cases. The arrow in the figure points out that CSP wins when the array size is around 65536 or larger.

7 Conclusion

Design rule checking is crucial in physical verification. As the size of modern VLSI circuits continues to grow, the demand for parallel, hardware-friendly DRC algorithms have been highlighted. In this paper, we have proposed to utilize a parallel sweepline algorithmic paradigm to solve a series of DRC problems. We have analyzed the theoretical complexity of the algorithms, implemented them on GPUs, and further integrated them into an end-to-end DRC flow. We conducted thorough experiments to demonstrate the effectiveness of the algorithms: they have achieved an average of 1.09×, 61.36×, and 45.00× speedup in three different DRC tasks, compared with a multi-threaded CPU design rule checker. We also provided other experimental results for further discussion.

In the future, we would like to investigate parallelizing the *merge* procedure with the sweepline paradigm, as it appears to be the new runtime bottleneck. Besides, we feel it necessary to develop more programming infrastructures for GPUs, including dynamic vectors, associative data structures, and their thread-safe solutions.

Acknowledgments

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

References

[1] J. L. Bentley and D. Wood, "An optimal worst case algorithm for reporting intersections of rectangles," *IEEE Transactions on Computers*, vol. 29, no. 07, pp. 571–577, 1980.

[2] U. Lauther, "An o (n log n) algorithm for boolean mask operations," in *Proc. DAC*, 1981, p. 555–562.

[3] M. Sato, J. Kim, T. Awashima, and T. Ohtsuki, "A theoretically optimal and practically fast algorithm for vlsi geometrical design rule verification," in *Proc. ISCAS*, 1988, pp. 1445–1448.

[4] C. R. Bonapace and C.-Y. Lo, "An o (n log m) algorithm for vlsi design rule checking," *IEEE TCAD*, vol. 11, no. 6, pp. 753–758, 1992.

[5] G. E. Bier and A. R. Pleszkun, "An algorithm for design rule checking on a multiprocessor," in *Proc. DAC*, 1985, pp. 299–304.

[6] F. Gregoretti and Z. Segall, "Analysis and evaluation of vlsi design rule checking implementation in a multiprocessor," in *Proc. Int. Conf. Parallel Processing*, 1984, pp. 7–14.

[7] 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, p. 182–187.

[8] E. C. Carlson and R. A. Rutenbar, "Mask verification on the connection machine," in *25th ACM/IEEE, Design Automation Conference. Proceedings 1988.* IEEE, 1988, pp. 134–140.

[9] ——, "Design and performance evaluation of new massively parallel vlsi mask verification algorithms in jigsaw," in *27th ACM/IEEE Design Automation Conference.* IEEE, 1990, pp. 253–259.

[10] J. D. Marantz, "Exploiting parallelism in vlsi cad," *Proc. IEEE ICCD'86*, 1986.

[11] K. MacPherson, "Parallel algorithms for layout verification," Master's thesis, Citeseer, 1995.

[12] G. Guo, T.-W. Huang, Y. Lin, and M. Wong, "Gpu-accelerated path-based timing analysis," in *2021 58th ACM/IEEE Design Automation Conference (DAC).* IEEE, 2021, pp. 721–726.

[13] Y. Lin, Z. Jiang, J. Gu, W. Li, S. Dhar, H. Ren, B. Khailany, and D. Z. Pan, "Dreamplace: Deep learning toolkit-enabled gpu acceleration for modern vlsi placement," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 40, no. 4, pp. 748–761, 2020.

[14] Y. Zhang, H. Ren, A. Sridharan, and B. Khailany, "Gatspi: Gpu accelerated gate-level simulation for power improvement," *arXiv preprint arXiv:2203.06117*, 2022.

[15] S. Liu, P. Liao, R. Zhang, Z. Chen, W. Lv, Y. Lin, and B. Yu, "Fastgr: Global routing on cpu-gpu with heterogeneous task graph scheduler," *IEEE/ACM Proceedings Design, Automation and Test in Europe (DATE)*, 2022.

[16] T.-W. Huang, D.-L. Lin, C.-X. Lin, and Y. Lin, "Taskflow: A lightweight parallel and heterogeneous task graph computing system," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 6, pp. 1303–1320, 2021.

[17] Z. Guo, J. Mai, and Y. Lin, "Ultrafast cpu/gpu kernels for density accumulation in placement," in *2021 58th ACM/IEEE Design Automation Conference (DAC).* IEEE, 2021, pp. 1123–1128.

[18] S. Lin, J. Liu, and M. D. Wong, "Gamer: Gpu accelerated maze routing," in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD).* IEEE, 2021, pp. 1–8.

[19] Z. Guo, T.-W. Huang, and Y. Lin, "Gpu-accelerated static timing analysis," in *Proceedings of the 39th International Conference on Computer-Aided Design*, 2020, pp. 1–9.

[20] Y. Lin, "Gpu acceleration in vlsi back-end design: Overview and case studies," in *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD).* IEEE, 2020, pp. 1–4.

[21] Y. Zhang, H. Ren, B. Keller, and B. Khailany, "Problem c: Gpu accelerated logic resimulation," in *Proceedings of the 39th International Conference on Computer-Aided Design*, 2020, pp. 1–4.

[22] G. Pasandi, S. Pratty, D. Brown, Y. Zhang, H. Ren, and B. Khailany, "2021 iccad cad contest problem c: Gpu accelerated logic rewriting," in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD).* IEEE, 2021, pp. 1–6.

[23] "KLayout," https://klayout.de/.

[24] J. JéJé, "An introduction to parallel algorithms," *Reading, MA: Addison-Wesley*, vol. 10, p. 133889, 1992.

[25] M. I. Shamos and D. Hoey, "Geometric intersection problems," in *17th Annual Symposium on Foundations of Computer Science (sfcs 1976).* IEEE, 1976, pp. 208–215.

[26] Y. Sun and G. E. Blelloch, "Parallel range, segment and rectangle queries with augmented maps," in *2019 Proceedings of the Twenty-First Workshop on Algorithm Engineering and Experiments (ALENEX).* SIAM, 2019, pp. 159–173.

[27] T. Ajayi, V. A. Chhabria, M. Fogaça, S. Hashemi, A. Hosny, A. B. Kahng, M. Kim, J. Lee, U. Mallappa, M. Neseem *et al.*, "Toward an open-source digital flow: First learnings from the openroad project," in *Proc. DAC*, 2019, pp. 1–4.