OpenDRC: An Efficient Open-Source Design Rule Checking Engine with Hierarchical GPU Acceleration

Zhuolun He[1,2], Yihang Zuo[2], Jiaxi Jiang[2], Haisheng Zheng[2], Yuzhe Ma[3], Bei Yu[1]
[1]CUHK [2]Shanghai AI Lab [3]HKUST(GZ)

2023 60th ACM/IEEE Design Automation Conference (DAC) | 979-8-3503-2348-1/23/$31.00 ©2023 IEEE | DOI: 10.1109/DAC56929.2023.10247734

***Abstract*—Design rule checking (DRC) is an essential procedure in physical verification, yet few open-source DRC tools are accessible in academia. To fill in the gap, we present OpenDRC, an open-source DRC engine that aims for extremely high efficiency. OpenDRC maintains hierarchical layouts with layer-wise bounding volume hierarchies and performs adaptive row-based partition to identify independent regions for check pruning and/or parallel processing. For common design rules, OpenDRC provides a sequential mode that runs cell-level sweeplines, and a parallel mode that launches edge-based GPU check kernels. Experiments demonstrate that OpenDRC outperforms state-of-the-art multi-threading and GPU-accelerated design rule checkers. The source code is available at https://github.com/opendrc/opendrc.**

I. Introduction

Design rule checking (DRC) is a critical stage in VLSI design flow that ensures a layout satisfies a deck of design rules imposed by process technology. Modern design rules consist of complex geometric constraints, such as constraints on distance, area, alignment, shape, and so on. Moreover, these rules may involve interactions between layers (e.g., constraints on the `NOT CUT` result between layers, minimum overlapping area constraints), as well as conditional rules (e.g., different spacing constraints given different projection lengths). The facts above, together with the explosion in the number of design rules as technology nodes scale down, have pushed DRC to become one of the most time-consuming stages in the whole design flow.

Improving tool/algorithm efficiency has been a central issue in the development of electronic design automation (EDA), which has received long-standing attention and abundant efforts. From the methodology perspective, these attempts could be classified into three categories, namely 1) to design better algorithms, 2) to parallelize computation workloads, and 3) to approximate desired results. **Designing better core algorithms** directly targets the underlying problem, which often has great impacts on the community. As for DRC, many theoretical results are obtained a few decades ago, such as rectangle intersection report [1], orthogonal range query [2], and boolean mask operations [3]. Proper data structures to cope with layout data are also discussed, including binary space partitioning data structures like quad-tree [4] and kd-tree [5], and hierarchies of bounding volumes like r-tree [6] and its variants. These historic milestones form the algorithmic foundations of today's design rule checkers. **Parallel computing** carries out computation workloads in multiple processors simultaneously to reduce turnaround time. Various kinds of data parallelism have been exploited: region-based methods [7], [8] partition the circuit into subregions for spatial parallelism; design hierarchy [9], [10] helps to realize cell-level parallelism; edge-based approaches [11], [12] increase parallelism within sweepline algorithms. Different design rules can be checked concurrently [13], attaining task parallelism, which could be further combined with data parallelism [14]. These works gain benefits from different hardware platforms, including SIMD engines [15], GPU [12], specialized hardware [16], [17], and distributed systems [7], [18]. **Approximation methods** sacrifice result accuracy to trade for improved runtime efficiency, among which machine learning (ML) algorithms are a popular subset. By predicting design rule violation (DRV) types of clipped layout regions, ML-based design rule checkers have demonstrated tens of times speedup compared with an accurate checker [19]. In the design stages (e.g., in placement or routing), ML is widely used to predict DRC hotspots [20] and the number of DRVs [21], without locating and identifying exact violations [19]. Although not directly accelerating the checking process, some ML-enhanced DRC schemes are worth mentioning, such as design rule verification [22], design rule augmentation [23], and DRC script generation [24].

Meanwhile, open-source EDA tools have been inspiring and empowering the evolution of cutting-edge EDA research. Many remarkable research outcomes would not be possible without the existence of public pioneering tools ABC [25], FLUTE [26], OpenTimer [27], DreamPlace [28], OpenROAD [29], etc. An open-source EDA tool facilitates researchers by 1) offering an off-the-shelf working solution to complete specific tasks, 2) serving as a strong baseline for algorithm development, and 3) providing infrastructures for data collection and golden result acquiring for ML applications. When it comes to 'design rule checking' in the literature, detailed routers (e.g., TritonRoute [30]) and layout editors (e.g., Magic [31], KLayout [32]) often integrate design rule checkers. Although basic design rule checking algorithms are implemented within these tools, they are not designed solely for physical verification purposes: detailed routers handle fundamental 'design rules' like short, spacing, and minimum area [33], while they are tightly coupled with the path search algorithms; layout viewers/editors are graphical user interface (GUI) centric, which are not optimized for standalone design rule checking. As design rule checking is a critical stage where many interesting research and design problems remain unsolved, we feel that a new design rule checking engine is necessary to support all these explorations. To this end, we propose OpenDRC, an open-source design rule checking engine that aims for extremely high efficiency. OpenDRC provides both sequential and parallel modes, where data structures and algorithms are accordingly customized for the different computational models. Besides, OpenDRC supports hierarchical layouts with layer-wise bounding volume hierarchies and is able to perform adaptive row-based layout partitioning to identify independent regions.

Our contributions are as follows: **(1)** we develop a new open-source design rule checking engine; **(2)** we come up with an adaptive row-based layout partition strategy that effectively generates independent clips for check pruning and parallel processing; **(3)** we present and implement customized data structures and algorithms for sequential (CPU) and parallel (GPU) hierarchical design rule checking; **(4)** we achieve a significant speedup of various design rule checks compared with state-of-the-art multithreading and GPU design rule checkers.

II. Preliminary

Design Rule. Design rules consist of geometric constraints imposed by specific fabrication technologies to achieve a high yield. Distance

This work is partially supported by The Research Grants Council of Hong Kong SAR (CUHK14208021) and National Key R&D Program of China (2020YFA0711900, 2020YFA0711903).

Authorized licensed use limited to: The Hong Kong University of Science and Technology (Guangzhou). Downloaded on May 09,2024 at 13:28:36 UTC from IEEE Xplore. Restrictions apply.

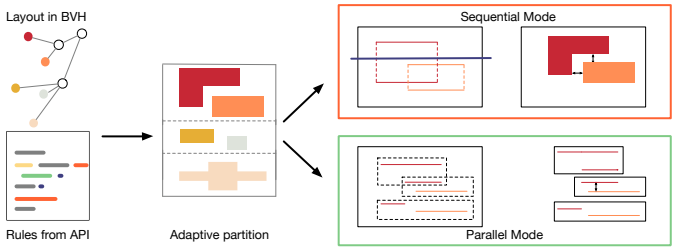

Fig. 1 The overall flow of OpenDRC.

constraints are the most common constraints, which include, depending on the positional relation between objects, width rules, spacing rules, extension rules, enclosure rules, and so on. Distance rules usually require a *minimum* distance between polygon edges due to various reasons [34]: the minimum width of polygons is limited by the resolution of the lithographic technique used, the minimum spacing between polygons is to ensure electrical isolation, and the minimum enclosure is to avoid layer misalignment errors. Other popular rules are *minimum area* rules, *shape* constraints (e.g., rectilinear), and *multi-color* design rules for multi-patterning lithography.

General-Purpose GPU and CUDA. The prosperous development of artificial intelligence has also popularized the concept of general-purpose graphics processing unit (GPGPU), which runs general-purpose programs on the hardware architecture initially dedicated to graphics rendering. GPGPUs offer massive computing power for highly parallel applications in various disciplines, which finds orders of magnitude performance gain. The programming model for GPGPUs is best described as Single Program Multiple Data (SPMD), where many parallel processing elements execute a single program on different input data, making them a good fit for data parallelism.

To enhance GPU programmability, higher-level programming environments have emerged, such as CUDA [35], OpenCL [36], and OpenACC [37]. CUDA comes with a software stack that extends C++ as the programming interface, attracting great attention in academia and industry. CUDA offers a thread hierarchy to organize parallel threads, which form one-, two-, or three-dimensional *thread blocks* [38]. *Thread blocks* are similarly organized into *block grids*. To allocate computation onto the above threads, CUDA defines *kernel* with an execution parameter N, which will be launched N times in N different CUDA threads. Each such thread is given a unique thread ID accessible with built-in variables in the *kernel*.

III. Overall Flow

We first introduce the overall flow of OpenDRC, as illustrated in Fig. 1. Given a hierarchical layout, OpenDRC parses the input file, and maintains the components in a layer-wise bounding volume hierarchy tree (detailed in Section IV-A). Meanwhile, design rules are specified from the provided programming interface (introduced in Section V-B). For the layers relevant to the specified design rules, OpenDRC performs an adaptive row-based partition of the layout, which effectively identifies independent regions (detailed in Section IV-B). After layout partitioning, OpenDRC provides a sequential (CPU) branch (detailed in Section IV-D) and a parallel (GPU) branch (detailed in Section IV-E) to execute the design rule checks.

IV. Algorithms

A. Layer-wise Bounding Volume Hierarchy

Hierarchical modularity is a natural solution for designers to cope with very large-scale systems. In the `GDSII` stream format [39], infinitely many hierarchical layers could be defined by recursive structure reference:

```
<structure> ::= BGNSTR STRNAME {<element>}* ENDSTR
<element>   ::= { ··· | <SREF> | ···} ··· ENDEL
<SREF>      ::= SREF ··· SNAME ···
```

In the above Backus Naur representation of the stream syntax, a ⟨`structure`⟩ is composed of a list of ⟨`elements`⟩, and an ⟨`element`⟩ could be, among others, a structure reference ⟨`SREF`⟩ that instantiates another structure defined elsewhere. Hereafter we use 'cell' and 'structure' interchangeably. To enable hierarchical design rule checking, OpenDRC does not flatten the layout, but preserves the layout hierarchy instead. Specifically, a structure reference effectively stores a *pointer* to the structure definition to reduce memory consumption.

One drawback of the layout hierarchy is that objects belonging to the same layer could scatter around the hierarchy tree. However, (range) queries for layer objects are very common since many design rules are defined for specific layers. To improve efficiency for such queries, OpenDRC maintains the *minimum bounding rectangle* (MBR) of each cell; for a cell that spans multiple layers, separated MBRs are computed for each layer and maintained. To answer a layer range query, it suffices to descend the hierarchy tree from the topmost ⟨`structure`⟩ (root), and prunes the whole subtree rooted at an element if its MBR for the interested layer is empty. Augmenting the hierarchy tree with MBRs reduces the layer range query complexity from $O(n)$ to $O(\min(n, kh))$, where n is the number of leaf nodes, k is the number of output, and h is the height of the hierarchy tree. Note that such MBR technique is widely applied in geometric data structures such as kd-trees [5] and R-trees [6].

Duplication and inverted indices. An effective strategy to trade space consumption for speed is to duplicate the hierarchy tree in a layer-wise manner. Namely, a separated hierarchy tree is built for each layer such that only modules containing objects in that layer are added to this hierarchy tree. The space consumption could be enlarged by at most L times where L is the number of layers. Suppose queries only ask for *all* objects in the given layer, it is possible to further construct element-level inverted indices that each contain a full list of leaf elements belonging to a layer.

B. Adaptive Row-based Partition

OpenDRC offers an adaptive row-based partition scheme that turns out to be very effective for check pruning and parallelization. The rationale behind is related to the popular row-based placement [28], [40]. The intuitions are twofold:

1) Layouts can be partitioned into non-overlapping regions (rows) along the y-axis, where cells do not overlap too much;
2) By grouping cells into independent rows, x-coordinates of cells in a row are more likely to be separated as well.

Technically, the row-based partition can be regarded as an interval merging problem, which can be efficiently solved in $\Theta(k + N)$ time, where k is the number of merge operations, and N is the size of the domain. In our case, k equals the number of cells, and N is the number of unique y-coordinates (discretization assumed). The algorithm can be divided into three steps: 1) initialize an array A of size N with indices as entry values; 2) merge y-coordinates belonging to the same cell; and 3) scan the whole array A to obtain the cover. To be specific, we use an '*pigeonhole* array' (of domain size N) to maintain the right endpoints of intervals, while interval left endpoints are indicated by the array indices. For each merge, only one array entry is updated in constant time. Algorithm 1 describes the details.

Note that the interval merging problem can also be solved without using the large *pigeonhole array* by sorting the merge targets, which yields an algorithm with time complexity $\Omega(k \log k)$. We come up with our solution since k is typically much larger than N in our problems, and arrays usually have a much better locality.

Authorized licensed use limited to: The Hong Kong University of Science and Technology (Guangzhou). Downloaded on May 09,2024 at 13:28:36 UTC from IEEE Xplore. Restrictions apply.

Algorithm 1 Interval Merging for Adaptive Layout Partition

Require: A set S of intervals to be merged
Ensure: Non-overlapping intervals covering the domain of S

```
Initialize an array A with indices                    ▷ Step1: Initialize
for all interval [l, r] ∈ S do                        ▷ Step2: Merge
    Update A[l] ← max(A[l], r)
end for
Initialize current interval end e ← −1
for the i-th element ∈ A do                           ▷ Step3: Scan
    if i > e then                          ▷ moving across interval boundary
        Create a new interval and reset e
    end if
    Update current interval end e ← max(e, A[i])
end for
```

C. Task Pruning from Hierarchy Tree

With the preserved hierarchy, OpenDRC always attempts to minimize the number of checks that are actually run. Redundancy could occur due to two possible reasons: 1) the check result could be inferred from previously finished checks; and 2) the check could be eliminated because violations *must* or *cannot* happen. In either the case, running an actual check is unnecessary. The former situation commonly occurs in hierarchical layouts, as they usually contain isomorphic modules that preserve geometric invariants under certain transformations, such as reflection and rotation, and under instantiation constructs like ⟨SREF⟩ and ⟨AREF⟩ in GDSII files. The latter situation can also be improved with the MBR augmented hierarchy tree.

Intra-Polygon Checks. As the finest granularity for transformations is usually at the polygon-level, there exist great optimization opportunities for intra-polygon checks. Given an intra-polygon check for a certain layer, OpenDRC performs *depth-first search* (DFS) along the hierarchy tree to locate layer objects. When a specific layer polygon is first encountered, the corresponding check is scheduled to the task graph. If the check is done for a leaf object, a tag is marked to indicate the finished check type. The same tag is marked for a non-leaf module if all submodules and leaf elements belonging to the module have been checked. In this way, OpenDRC tries to reuse check results when visiting a cell reference element: if the corresponding cell has already been checked elsewhere, and the transformations preserve the target properties of the check, the check result could be safely reused.

Inter-Polygon Checks. Inter-polygon checks are slightly more complicated as many invariants are no longer preserved under common constructs. Nevertheless, OpenDRC still attempts to explore opportunities to reduce workloads. Given an inter-polygon check between layer M and layer N (M and N could be identical), OpenDRC still searches along the hierarchy tree from the root, denoted as a pair of nodes $(root^M, root^N)$. A similar memoization strategy is used as described in intra-polygon checks. Specifically, only if (a^M, a^N) has been checked, OpenDRC marks it down for possible reuse. Note that the check result of (a^M, b^N) cannot be reused if a and b do not belong to the same parent cell, because another instantiation of them may not be of the same relative position. A check for node pair (a^M, b^N) could possibly be eliminated if:

- $M = N \wedge id_a > id_b$. Node id assignment could be arbitrary. This is a duplication of the check for (b^M, a^N).
- $a = b$ and (a^M, a^N) has been checked. This corresponds to redundancy case 1) we described.
- $\text{MBR}_a^M \cap \text{MBR}_b^N = \emptyset$. This corresponds to redundancy case 2) we described.

Technically, the MBRs should be enlarged by a minimum rule distance to ensure non-overlapping indeed indicates no violations.

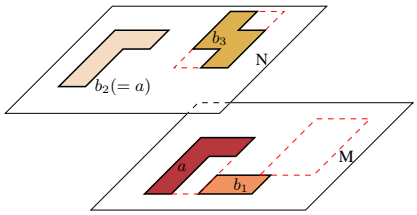

Fig. 2 Three inter-polygon checks could be eliminated.

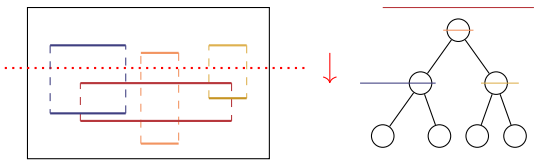
Fig. 3 Sweepline and interval tree for overlapping MBR query.

Fig. 2 illustrates the above three cases. When $M = N$, the check (a^M, b_1^N) is a duplication of (b_1^M, a^N). As b_2 and a refer to the same cell, the check result of (a^M, a^N) can be reused if it is already checked. The check (a^M, b_3^N) can be pruned because their MBRs are non-overlapping.

D. The Sequential Mode

As by the task pruning strategy introduced in Section IV-C, the sequential mode of OpenDRC first detects potential violations between objects by querying overlapping MBRs of polygons or cells, and then performs edge-based checks among those object pairs.

Overlapping MBR Query. OpenDRC runs a standard sweepline algorithm [1] to detect all overlapping MBRs, except that interval trees [41] are used instead of segment trees for implementation simplicity. An interval tree is a binary search tree that stores an interval I in the highest node satisfying $u \in I$, where u is the key of this node. Specifically, every node of the interval tree maintains its intervals in two separate lists: one is sorted by left endpoints, and the other is sorted by right endpoints. By the definitions above, all left endpoints (resp. right endpoints) stored in the right (resp. left) subtree are larger (resp. smaller) than the parent node's key, which enables efficient range queries. The sweepline algorithm moves a conceptual line across the plane from top to bottom, which scans through the top and bottom sides of all MBRs in descending y. When the top side of an MBR m is encountered, the corresponding horizontal interval is inserted into the interval tree, and a query to the interval tree reports all the MBRs overlapping with m. When the bottom side of m is encountered, the horizontal interval is removed from the interval tree. Fig. 3 illustrates the sweepline procedure and the corresponding interval tree.

Check Procedures. For distance rules, edge-to-edge checks need to be performed, be it an intra-polygon check extracted from the hierarchy tree, or an inter-polygon check obtained from MBR queries. Polygon vertices are stored in clockwise order, so that positional relations of edges are determined accordingly. For area rule checks, OpenDRC computes polygon areas by the *Shoelace Theorem*.

E. The Parallel Mode

The parallel mode of OpenDRC runs design rule checks on GPUs, which utilizes very different algorithms and data structures from those for sequential processing. After layout partitioning, OpenDRC performs parallel design rule checks in a row-by-row manner, as cells belonging to different rows will not produce any violation.

Before checking, OpenDRC packs the edges of relevant polygons into a flattened array, which is transferred from the host memory to the

Authorized licensed use limited to: The Hong Kong University of Science and Technology (Guangzhou). Downloaded on May 09,2024 at 13:28:36 UTC from IEEE Xplore. Restrictions apply.

GPU device memory. Depending on the complexity of each polygon or polygon pair, OpenDRC selects either a brute-force executor or a sweepline executor. For smaller tasks, parallel threads are launched for each polygon (or pair), in which edge pairs are enumerated and checked. For larger tasks, a parallel sweepline algorithm is performed, which is similar to the one described in X-Check [12]: firstly, a parallel *scan* determines the check range of each edge; then parallel threads are launched to perform the check between an edge and all other edges within its check range. Although these two steps can be combined theoretically, separating them into two kernel launches enables efficient kernel code optimization (viz. `for` loops versus `while` loops).

V. Design and Implementation Details

A. Software Architecture

Conceptually, OpenDRC consists of four layers, from topmost to bottommost:

1) the `interface` layer,
2) the `application` layer,
3) the `algorithm` layer, and
4) the `infrastructure` layer.

In general, higher layers depend on the abstraction of lower layers, but not the other way around. The `interface` layer is responsible for the interaction between OpenDRC and the outside world, such as reading design files, defining rule decks, adaptors to design databases, and result output. The `application` layer can be regarded as a system controller that schedules computation tasks and dispatches them to algorithms. The `algorithm` layer, as indicated by the name, consists of the implementation of design rule checking algorithms, such as `width-check` and `space-check`. The `infrastructure` layer is for abstract data structure and algorithms, various program utilities (timer, logger, etc.), and some basic GPU libraries.

B. General Programming Interface

OpenDRC aims to provide extreme extensibility and usability through its general programming interface. OpenDRC recognises the need of researchers and end users to customize their usage of the engine, so it encourages the use of the C++ programming interface, instead of another scripting language, as the default way to define design rule checking tasks. The code snippet in Listing 1 demonstrates how to

Listing 1 Code snippet of using OpenDRC.

```
auto db = odrc::gdsii::read(/* path-to-gdsii */);
auto e  = odrc::engine();
e.add_rules({
  db.polygons().is_rectilinear(),
  db.layer(19).width().greater_than(18),
  db.layer(20).polygons().ensures(
      [](const auto& p){return !p.name.empty();}
  )
});
e.check(db);
```

program OpenDRC. We start by reading-in a layout file and creating an instance of the DRC engine. Then we specify a list of design rules using the `add_rules` method, where each rule is described in *chaining methods* that resemble natural language. In this example, we have defined three rules: the first rule ensures that all the polygons are rectilinear; the second rule ensures the minimal width in layer 19 is 18nm; the third rule ensures that every polygon in layer 20 has a non-empty name. Finally, calling `check()` will run checks for the specified rules.

OpenDRC defines two categories of methods, *selectors* and *predicates* to help define design rules. Selectors locate the target objects for which a design rule is defined. In our example, chained methods `layer(19).width()` selects the width in the 19-th layer as the check target. Predicates are the conditions that the selected objects need to conform to, such as `is_rectilinear()` that requires axis-aligned shapes. Specifically, the `ensures()` method takes a callable as a parameter that enables user-defined predicates.

C. Heterogeneous Computing via Asynchronous Operations

A CPU-GPU computing platform is heterogeneous, which requires special considerations on the orchestration between them. OpenDRC utilizes asynchronous operations and Stream Ordered Memory Allocator [38] to hide communication or computation latencies. When OpenDRC starts, it creates CUDA stream objects that are responsible for asynchronous operations. As the parsing is finished and the database is ready, asynchronous data copies are launched to prepare necessary data (e.g., polygon edges) for parallel checks. The data movement is thus usually hidden by the layout partitioning in the flow. OpenDRC also tries to overlap CPU computation and GPU processing to hide latency. For example, parallel checks of a row (taking place on device) can be performed concurrently with the necessary data preprocessing of the next row (taking place on host).

D. Functors and Type Traits

The extensibility of OpenDRC also comes from a generic implementation of underlying functors. The sweepline functor shown in Listing 2 is a typical example, which is regarded as a metafunction that takes another callable as a parameter. Here an executor is either a wrapper for

Listing 2 The sweepline functor.

```
template <typename Executor, typename EventIt,
          typename Status,   typename Op>
void sweepline(Executor&& exec, EventIt first,
               EventIt last, Status* st, Op op) {
  if constexpr (std::is_same_v<std::remove_cv_t<
    std::remove_reference_t<decltype(exec)>>,
    odrc::execution::sequenced_policy>) { // CPU
  } else { /* GPU */ }
}
```

a CUDA stream object (`cudaStream_t`), indicating the operation will be appended to the stream, or a simple `odrc::sequenced_policy` object that indicates a sequential operation.

OpenDRC utilizes type traits to manage certain properties of rules and checks, which dispatches function calls at compile time and avoids runtime branching. Line 5-8 in Listing 2 demonstrates how OpenDRC decides whether a sweepline operates on CPU or GPU by accessing the type traits of the executor in a *constexpr if statement*. Another typical usage is to mark the rule types by the target edge relations (e.g., width or space), which is also implemented in KLayout [32] as runtime arguments. In general, using type traits slightly improves runtime efficiency and helps organize code logic concisely.

VI. Experimental Evaluation

OpenDRC is implemented in C++17 and CUDA. Experiments were conducted on a Linux machine with an Intel Core i7-11700 processor (2.5GHz), 64GB main memory, and an NVIDIA GeForce GTX 1660Ti graphics card. Benchmark layouts are synthesized from OpenROAD [29], with the ASAP7 [42] process design kit (PDK) and all default settings provided in the flow scripts.

To evaluate the efficiency of OpenDRC, we compare its performance with the state-of-the-art multi-threading design rule checker, *KLayout* [32], and the state-of-the-art GPU design rule checker, *X-Check* [12]. KLayout provides three different operation modes, namely *flat* mode, *deep* (hierarchy) mode, and *tiling* mode. In the deep mode, the operations will be performed in a hierarchical fashion; in the tiling mode, operations are evaluated in tiles, and multi-CPU support is

Authorized licensed use limited to: The Hong Kong University of Science and Technology (Guangzhou). Downloaded on May 09,2024 at 13:28:36 UTC from IEEE Xplore. Restrictions apply.

TABLE I Runtime comparisons for intra-polygon design rule checks.

Design	Rule	KLayout			X-Check	OpenDRC		Rule	KLayout			X-Check	OpenDRC	
		flat	deep	tile		Seq.	Par.		flat	deep	tile		Seq.	Par.
aes	M1.W.1	3.45	12.69	0.49	0.41	0.02	0.03	M1.A.1	3.34	3.32	0.65	-	0.02	0.03
	M2.W.1	1.37	3.83	0.23	0.14	0.04	0.04	M2.A.1	1.35	1.33	0.37	-	0.04	0.04
	M3.W.1	2.52	2.98	0.36	0.11	0.03	0.03	M3.A.1	2.49	2.51	0.51	-	0.03	0.03
ethmac	M1.W.1	11.88	45.84	1.56	1.21	0.07	0.08	M1.A.1	11.55	11.55	2.05	-	0.07	0.08
	M2.W.1	3.76	10.72	0.52	0.42	0.10	0.11	M2.A.1	3.62	3.63	1.01	-	0.10	0.11
	M3.W.1	6.36	7.64	0.77	0.31	0.08	0.08	M3.A.1	6.20	6.24	1.24	-	0.08	0.08
gcd	M1.W.1	0.13	0.44	0.13	0.11	< 0.01	< 0.01	M1.A.1	0.13	0.13	0.13	-	< 0.01	< 0.01
	M2.W.1	0.05	0.08	0.05	< 0.01	< 0.01	< 0.01	M2.A.1	0.05	0.05	0.05	-	< 0.01	< 0.01
	M3.W.1	0.06	0.07	0.06	< 0.01	< 0.01	< 0.01	M3.A.1	0.06	0.06	0.06	-	< 0.01	< 0.01
ibex	M1.W.1	3.60	12.38	0.50	0.43	0.02	0.03	M1.A.1	3.52	3.52	0.65	-	0.02	0.03
	M2.W.1	1.30	3.61	0.24	0.14	0.03	0.04	M2.A.1	1.27	1.28	0.36	-	0.04	0.04
	M3.W.1	2.38	2.88	0.36	0.10	0.03	0.03	M3.A.1	2.36	2.35	0.51	-	0.03	0.03
jpeg	M1.W.1	13.32	55.35	1.68	1.39	0.08	0.08	M1.A.1	13.01	13.00	2.17	-	0.07	0.08
	M2.W.1	3.05	8.77	0.46	0.40	0.10	0.10	M2.A.1	2.98	2.95	0.95	-	0.09	0.09
	M3.W.1	4.86	6.14	0.59	0.29	0.08	0.08	M3.A.1	4.79	4.81	1.10	-	0.08	0.07
sha3	M1.W.1	3.48	12.36	0.49	0.43	0.02	0.03	M1.A.1	3.40	3.40	0.63	-	0.02	0.03
	M2.W.1	1.10	2.95	0.21	0.12	0.03	0.03	M2.A.1	1.07	1.09	0.33	-	0.03	0.03
	M3.W.1	1.79	2.15	0.30	0.09	0.02	0.02	M3.A.1	1.79	1.77	0.42	-	0.02	0.02
uart	M1.W.1	0.15	0.40	0.15	0.11	< 0.01	< 0.01	M1.A.1	0.14	0.14	0.15	-	< 0.01	< 0.01
	M2.W.1	0.06	0.12	0.06	< 0.01	< 0.01	< 0.01	M2.A.1	0.06	0.06	0.06	-	< 0.01	< 0.01
	M3.W.1	0.08	0.09	0.08	< 0.01	< 0.01	< 0.01	M3.A.1	0.08	0.08	0.08	-	< 0.01	< 0.01
Average		37.7×	82.1×	9.6×	4.5×	0.9×	1.0×		37.6×	37.6×	13.0×	-	1.0×	1.0×

enabled [32]. These three modes are exclusive, so we list DRC runtime under the three options in individual columns since no combination of them is directly accessible. We reimplement the vertical sweeping algorithm proposed in X-Check (Section 4.1 in their paper [12]).

We follow the experimental settings in X-Check [12] to check (minimum) *width*, *spacing*, and *enclosure* rules; we further implement *minimum area* checks that X-Check is unable to deal with. These rules are typical, as they include two intra-polygon rules (*width*, *area*) and two inter-polygon rules (*spacing*, *enclosure*); *enclosure* rules are inter-layer while others are intra-layer; except *area* rules, other rules are essentially *distance* rules. The selected rules involve Back-End-Of-Line (BEOL) layers M1, M2, M3, V1, and V2 [42].

Runtime comparisons for intra-polygon checks are shown in TABLE I, and comparisons for inter-polygon checks are in TABLE II. The 'average' rows are normalized against the parallel mode of OpenDRC, where the runtime is the *geometric mean* of the column, as we value all checks equally regardless of their sizes. Note again that X-Check is unable to perform area checks, so the column is empty. Intra-polygon checks generally run fast, which confirms the claim in X-Check [12]. OpenDRC achieves $4.5\times$ speedup on average compared with X-Check, and $9.6\times$ - $13.0\times$ speedup compared with KLayout (tiling mode). For sequential modes, OpenDRC is around $37.6\times$ faster than the flat/deep mode of KLayout, which we argue is due to the hierarchy strategy OpenDRC adopts. In fact, both sequential and parallel modes of OpenDRC run equally fast for intra-polygon checks. Inter-polygon checks have heavier computation workloads, where we see more significant speedup from GPU acceleration. For space checks, GPU-accelerated OpenDRC is $3.2\times$, $5.6\times$, and $12.0\times$ faster than sequential OpenDRC, X-Check, and KLayout (tiling mode), respectively; for enclosing checks, the speedups become $4.7\times$, $2.9\times$, $61.5\times$, respectively.[1] The sequential implementation of OpenDRC is also $14.9\times$ - $91.3\times$ faster than KLayout (the faster one in flat/deep mode). The experiments demonstrate the efficiency of OpenDRC and the effectiveness of the proposed techniques.

We also provide a runtime breakdown of OpenDRC in Fig. 4, taking sequential space checks as an example. Since asynchronous operations are utilized in the parallel mode, runtime profiling and visualization are slightly complicated and are left to future work. As can be seen,

[1]We notice the abnormal runtime of KLayout reported in X-Check [12], which we think could be due to a very large number of violations that trigger abnormal program behavior (e.g., heavy disk IO, etc.).

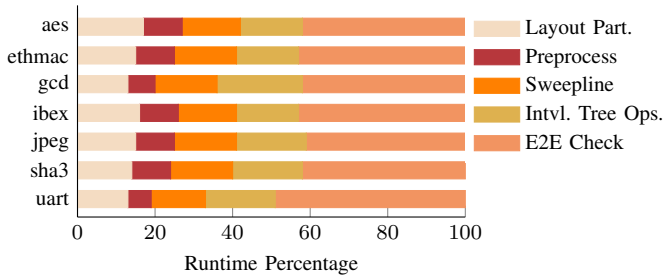

Fig. 4 The runtime breakdown of OpenDRC sequential *minimum spacing* checks. 'Layout Part' refers to adaptive layout partitioning; 'Intvl. Tree Ops.' refers to interval tree operations `insert`, `remove`, and `query`; 'E2E Check' refers to edge-to-edge checks.

the adaptive layout partition consumes only around 15% of overall runtime, but greatly enhances the efficiency of subsequent steps. The sweepline algorithm, together with operations in the interval tree, taking around 35% of runtime, examines overlapping of cell MBRs and prunes unnecessary checks. Finally, 40% - 50% of the overall runtime is spent on edge-to-edge space checks.

VII. Conclusion and Roadmap

As inspired by many interesting research problems in VLSI layout operations and design rule checking, we develop OpenDRC, a new open-source design rule checking engine. By introducing adaptive row-based layout partition and efficient sequential/parallel hierarchical DRC procedures, OpenDRC achieves significant speedup compared with state-of-the-art multi-threading and GPU design rule checkers. Ongoing works for OpenDRC include a systematic evaluation of heterogeneous computing in DRC, data compression techniques for memory footprint reduction, and supports for general geometric shapes.

References

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

[2] D. E. Willard, "New data structures for orthogonal range queries," *SIAM Journal on Computing (SICOMP)*, vol. 14, no. 1, pp. 232–253, 1985.

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

[4] R. A. Finkel and J. L. Bentley, "Quad trees a data structure for retrieval on composite keys," *Acta informatica*, vol. 4, no. 1, pp. 1–9, 1974.

Authorized licensed use limited to: The Hong Kong University of Science and Technology (Guangzhou). Downloaded on May 09,2024 at 13:28:36 UTC from IEEE Xplore. Restrictions apply.

TABLE II Runtime comparisons for inter-polygon design rule checks.

Design	Rule	KLayout flat	KLayout deep	KLayout tile	X-Check	OpenDRC Seq.	OpenDRC Par.	Rule	KLayout flat	KLayout deep	KLayout tile	X-Check	OpenDRC Seq.	OpenDRC Par.
aes	M1.S.1	4.33	13.78	0.62	0.17	0.21	0.06	V1.M1.EN.1	468.24	462.28	15.97	0.20	6.44	0.12
	M2.S.1	1.55	4.15	0.29	0.13	0.09	0.02	V2.M2.EN.1	2.93	1.64	0.59	0.14	0.18	0.09
	M3.S.1	2.64	3.25	0.38	0.12	0.15	0.02	V1.M2.EN.2	469.96	468.89	15.71	0.20	0.24	0.12
ethmac	M1.S.1	14.67	48.50	1.89	0.39	0.72	0.14	V1.M1.EN.1	3045.02	3038.10	57.76	2.00	42.35	0.41
	M2.S.1	4.35	11.71	0.59	0.20	0.23	0.05	V2.M2.EN.1	8.29	4.74	1.45	0.23	0.47	0.22
	M3.S.1	6.68	8.17	0.82	0.16	0.39	0.04	V1.M2.EN.2	3031.20	3034.67	55.63	0.36	0.84	0.32
gcd	M1.S.1	0.15	0.46	0.14	0.11	< 0.01	0.01	V1.M1.EN.1	3.06	2.96	3.09	0.11	0.06	< 0.01
	M2.S.1	0.05	0.09	0.05	0.11	< 0.01	< 0.01	V2.M2.EN.1	0.07	0.05	0.08	0.10	< 0.01	< 0.01
	M3.S.1	0.06	0.07	0.06	0.11	< 0.01	< 0.01	V1.M2.EN.2	2.95	2.95	2.99	0.10	< 0.01	< 0.01
ibex	M1.S.1	4.45	13.15	0.63	0.17	0.22	0.06	V1.M1.EN.1	477.86	473.62	16.03	0.21	7.14	0.13
	M2.S.1	1.49	3.96	0.25	0.13	0.09	0.02	V2.M2.EN.1	2.78	1.56	0.56	0.15	0.18	0.08
	M3.S.1	2.50	3.08	0.39	0.12	0.14	0.02	V1.M2.EN.2	479.79	477.17	15.90	0.17	0.24	0.12
jpeg	M1.S.1	15.82	57.36	2.01	0.43	0.80	0.16	V1.M1.EN.1	3609.55	3580.46	58.29	1.59	55.07	0.49
	M2.S.1	3.48	9.79	0.49	0.21	0.20	0.05	V2.M2.EN.1	7.07	4.04	1.22	0.22	0.40	0.20
	M3.S.1	5.17	6.70	0.64	0.16	0.30	0.03	V1.M2.EN.2	3611.69	3588.04	57.01	0.35	0.87	0.32
sha3	M1.S.1	4.23	13.02	0.60	0.16	0.21	0.06	V1.M1.EN.1	476.10	472.44	15.87	0.49	7.07	0.12
	M2.S.1	1.16	3.23	0.22	0.12	0.07	0.02	V2.M2.EN.1	2.32	1.29	0.48	0.13	0.14	0.07
	M3.S.1	1.87	2.31	0.30	0.11	0.11	0.02	V1.M2.EN.2	468.70	467.92	17.28	0.15	0.22	0.11
uart	M1.S.1	0.19	0.44	0.19	0.11	< 0.01	0.01	V1.M1.EN.1	3.61	3.50	3.62	0.10	0.06	< 0.01
	M2.S.1	0.07	0.13	0.07	0.11	< 0.01	< 0.01	V2.M2.EN.1	0.10	0.06	0.10	0.12	< 0.01	< 0.01
	M3.S.1	0.08	0.10	0.08	0.10	< 0.01	< 0.01	V1.M2.EN.2	3.49	3.48	3.54	0.10	< 0.01	< 0.01
Average		47.6×	99.5×	12.0×	5.6×	3.2×	1.0×		514.9×	429.0×	61.5×	2.9×	4.7×	1.0×

[5] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[6] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *Proc. SIGMOD*, 1984, pp. 47–57.

[7] S. Nandy, "Geometric Design Rule Check of VLSI Layouts in Distributed Computing Environment," *Proc. VLSI Design*, vol. 1, no. 2, pp. 155–167, 1994.

[8] K.-T. Hsu, S. Sinha, Y.-C. Pi, C. Chiang, and T.-Y. Ho, "A distributed algorithm for layout geometry operations," in *Proc. DAC*. IEEE, 2011, pp. 182–187.

[9] F. Gregoretti and Z. Segall, "Analysis and evaluation of parallel rectangle intersection for vlsi design rule checking," *Microprocessors and Microsystems*, vol. 19, no. 2, pp. 85–100, 1987.

[10] N. Hedenstierna and K. Jeppson, "A parallel hierarchical design rule checker," in *European Conference on Design Automation*. IEEE Computer Society, 1992, pp. 142–143.

[11] E. C. Carlson and R. A. Rutenbar, "Mask verification on the connection machine," in *Proc. DAC*. IEEE, 1988, pp. 134–140.

[12] Z. He, Y. Ma, and B. Yu, "X-Check: GPU-Accelerated Design Rule Checking via Parallel Sweepline Algorithms," in *Proc. ICCAD*, 2022.

[13] J. D. Marantz, "Exploiting parallelism in VLSI CAD," 1986.

[14] 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.

[15] S. Koranne, "A high performance SIMD framework for design rule checking on Sony's PlayStation 2 Emotion Engine platform [IC layout]," in *International Symposium on Signals, Circuits and Systems. Proceedings, SCS 2003.(Cat. No. 03EX720)*. IEEE, 2004, pp. 371–376.

[16] R. Kane and S. Sahni, "A systolic design rule checker," in *Proc. DAC*. IEEE, 1984, pp. 243–250.

[17] Z. Luo, M. Martonosi, and P. Ashar, "An edge-endpoint-based configurable hardware architecture for VLSI layout Design Rule Checking," *Proc. VLSI Design*, vol. 10, no. 3, pp. 249–263, 2000.

[18] A. Pais, M. Anido, and C. Oliveira, "Developing a distributed architecture for design rule checking," in *Proc. MWSCAS*, vol. 2, 2001, pp. 678–681.

[19] L. Francisco, T. Lagare, A. Jain, S. Chaudhary, M. Kulkarni, D. Sardana, W. R. Davis, and P. Franzon, "Design Rule Checking with a CNN Based Feature Extractor," in *Proc. MLCAD*. IEEE, 2020, pp. 9–14.

[20] W. Zeng, A. Davoodi, and R. O. Topaloglu, "Explainable DRC hotspot prediction with random forest and SHAP tree explainer," in *Proc. DATE*, 2020, pp. 1151–1156.

[21] Z. Xie, Y.-H. Huang, G.-Q. Fang, H. Ren, S.-Y. Fang, Y. Chen, and J. Hu, "RouteNet: Routability prediction for mixed-size designs using convolutional neural network," in *Proc. ICCAD*. IEEE, 2018, pp. 1–8.

[22] I. Alam, T. Li, S. Brock, and P. Gupta, "DRDebug: Automated Design Rule Debugging," *IEEE TCAD*, 2022.

[23] V. Dai, L. Capodieci, J. Yang, and N. Rodriguez, "Developing DRC Plus rules through 2D pattern extraction and clustering techniques," in *Proc. SPIE*, vol. 7275. SPIE, 2009, pp. 332–341.

[24] B. Zhu, X. Zhang, Y. Lin, B. Yu, and M. Wong, "Efficient design rule checking script generation via key information extraction," in *Proc. MLCAD*, 2022, pp. 77–82.

[25] R. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool," in *Proc. CAV*. Springer, 2010, pp. 24–40.

[26] C. Chu and Y.-C. Wong, "FLUTE: Fast lookup table based rectilinear Steiner minimal tree algorithm for VLSI design," *IEEE TCAD*, vol. 27, no. 1, pp. 70–83, 2007.

[27] T.-W. Huang and M. D. Wong, "OpenTimer: A high-performance timing analysis tool," in *Proc. ICCAD*, 2015, pp. 895–902.

[28] Y. Lin, S. Dhar, W. Li, H. Ren, B. Khailany, and D. Z. Pan, "Dreamplace: Deep learning toolkit-enabled gpu acceleration for modern vlsi placement," in *Proc. DAC*, 2019, pp. 1–6.

[29] 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.

[30] A. B. Kahng, L. Wang, and B. Xu, "TritonRoute: An initial detailed router for advanced VLSI technologies," in *Proc. ICCAD*, 2018, pp. 1–8.

[31] J. K. Ousterhout, G. T. Hamachi, R. N. Mayo, W. S. Scott, and G. S. Taylor, "The magic VLSI layout system," *IEEE MDAT*, vol. 2, no. 1, pp. 19–30, 1985.

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

[33] G. Chen, C.-W. Pui, H. Li, and E. F. Young, "Dr. CU: Detailed routing by sparse grid graph and minimum-area-captured path search," *IEEE TCAD*, vol. 39, no. 9, pp. 1902–1915, 2019.

[34] K. Bhanushali and W. R. Davis, "FreePDK15: An open-source predictive process design kit for 15nm FinFET technology," in *Proc. ISPD*, 2015, pp. 165–170.

[35] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with cuda: Is cuda the parallel programming model that application developers have been waiting for?" *Queue*, vol. 6, no. 2, pp. 40–53, 2008.

[36] A. Munshi, "The opencl specification," in *2009 IEEE Hot Chips 21 Symposium (HCS)*. IEEE, 2009, pp. 1–314.

[37] "The OpenACC Application Programming Interface," OpenACC-Standard.org, Specification, Nov. 2021, version 3.2.

[38] "CUDA C++ Programming Guide," https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[39] "GDSII Stream Format Manual," http://bitsavers.informatik.uni-stuttgart.de/pdf/calma/GDS_II_Stream_Format_Manual_6.0_Feb87.pdf.

[40] J. Lu, P. Chen, C.-C. Chang, L. Sha, D. J.-H. Huang, C.-C. Teng, and C.-K. Cheng, "eplace: Electrostatics based placement using nesterov's method," in *Proc. DAC*, 2014, pp. 1–6.

[41] E. M. McCreight, "Efficient algorithms for enumerating intersecting intervals and rectangles," Tech. Rep., 1980.

[42] L. T. Clark, V. Vashishtha, L. Shifren, A. Gujja, S. Sinha, B. Cline, C. Ramamurthy, and G. Yeric, "ASAP7: A 7-nm finFET predictive process design kit," *Microelectronics Journal*, vol. 53, pp. 105–115, 2016.

Authorized licensed use limited to: The Hong Kong University of Science and Technology (Guangzhou). Downloaded on May 09,2024 at 13:28:36 UTC from IEEE Xplore. Restrictions apply.