IEEE/ACM

2022 INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN

ICCAD

41st Edition

IEEE/ACM 2022 INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN ICCAD 41st Edition

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

Zhuolun He[1], Yuzhe Ma[2], Bei Yu[1]

[1] The Chinese University of Hong Kong
[2] HKUST(GZ)

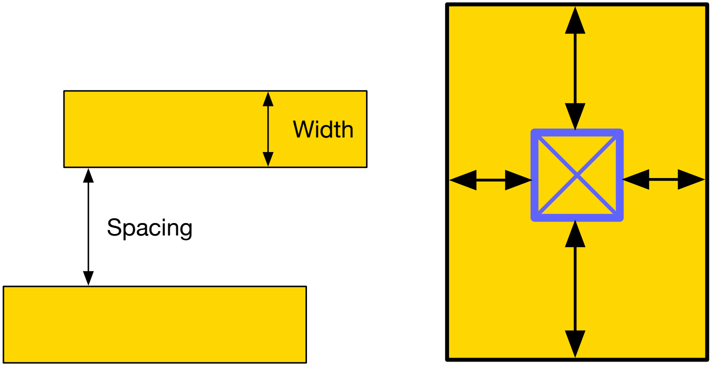
Sept. 14, 2022

Outline

Background and Motivation

Design Rule Checking (DRC)

DRC: to ensure the layout does not violate geometric constraints



(a) (b)

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

Why GPU-Accelerated DRC?

- Design rule number explosion in advanced technology
- Many classic parallel algorithms do not scale beyond a few CPU cores [G. Guo+, DAC'21]
  - data parallelism
  - task parallelism
- GPUs have demonstrated potential in EDA tool acceleration

Methodologies to Develop Efficient GPU-enabled EDA Tool

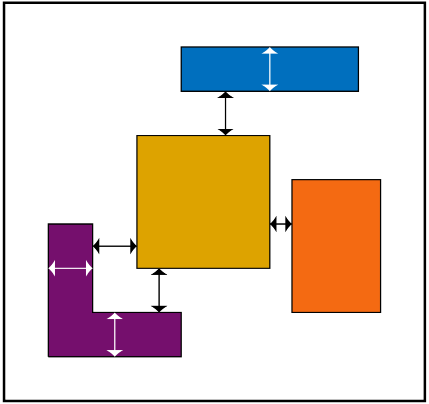
- to cast a design automation problem into another problem solvable by current tools/infrastructure
  - DreamPlace (analytical placement → NN training) [Y. Lin+, DAC'19]
  - GATSPI (gate-level simulation → graph manipulation) [Y. Zhang+, DAC'22]
  - FastGR (batched net routing ordering → task scheduling) [S. Liu+, DATE'22]
- to design novel GPU-friendly computation kernels for some critical tasks in the design flow
  - Placement [Z. Guo+, DAC'21]
  - GAMER (maze routing) [S. Lin+, ICCAD'21]
  - STA [Z. Guo+, ICCAD'20]

Our work is closer to the second methodology.

Problem Formulation (informal)

Problem (Distance Check (informal))

- *Layout: a set of axis-parallel polygonal objects*
- *Distance rule: any two edges must not be closer than a predefined minimal distance*
- *Distance violation: a pair of edges in the layout that violate the distance rule*
- *Our task: report all the distance violations*



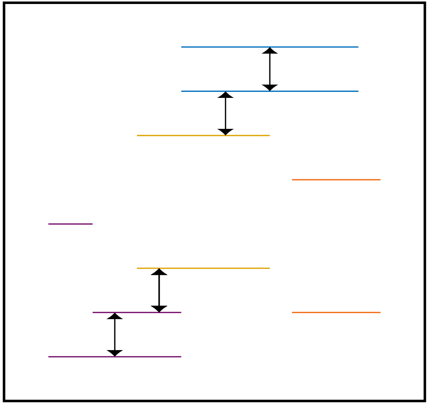
Problem Formulation

(We only consider horizontal edges.)

Problem (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:

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



Sequential Sweepline Algorithm for Distance Check

1. Sort segment endpoints $\mathcal{P}$ by ascending x-coordinates
2. Initialize an empty BST $\mathcal{S}$ (using y-coordinates as keys)
3. Scan endpoints from left to right
    1. If p is the left endpoint of a segment $h = [l, r] \times y$
        1. Range query $\mathcal{S}$ for $[y - \delta, y + \delta]$
        2. Report the corresponding segment pairs
        3. Insert h to $\mathcal{S}$
    2. Otherwise (i.e., right endpoint)
        1. Delete h from $\mathcal{S}$

Complexity: $O(n \log n + k)$, optimal:

- element uniqueness problem (lower bounded by $\Omega(n \log n)$) reducible to it
- we need $\Omega(k)$ time to report all the violations

Algorithm: Parallel Vertical Sweeping

Another View of Sweep Line

Prefix Structure

$$a[] = (4, 5, 3, 6, 2, 5, 1, 1, 0)$$

Prefix sums:

$$s = (4, 9, 12, 18, 20, 25, 26, 27, 27)$$

Can we do it in parallel?

Parallel prefix sums

$$a[] = (4, 5, 3, 6, 2, 5, 1, 1, 0)$$

Suppose we have 3 threads.

1. Batching: each thread computes sums of 3 consecutive elements.

$$s = (?, ?, 12, ?, ?, 13, ?, ?, 2)$$

2. Sweeping: sweep the partial sums

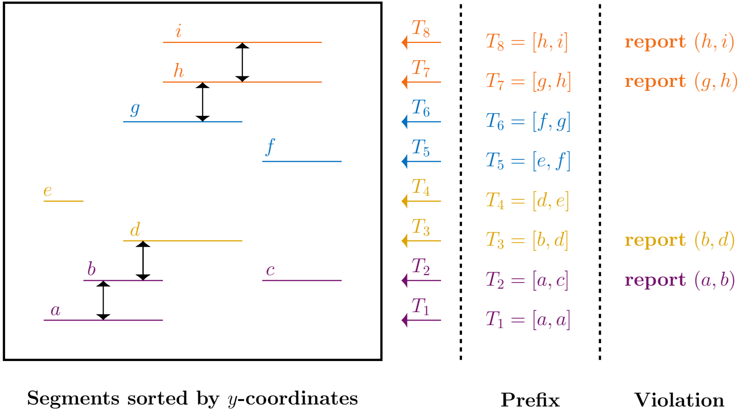
$$s = (?, ?, 12, ?, ?, 25, ?, ?, 27)$$

3. Refining: compute other prefix sums

$$s = (4, 9, 12, 18, 20, 25, 26, 27, 27)$$

Vertical Sweeping

- Key idea: the prefix structure contains a set $\mathcal{S}$ of segments that are below current segment within δ in y-direction
- Remains to check if each pair of segments overlap in the x-direction



Complexity Analysis

Assume we have n elements evenly distributed to b blocks.
Let s_i be the size of the i-th prefix structure.

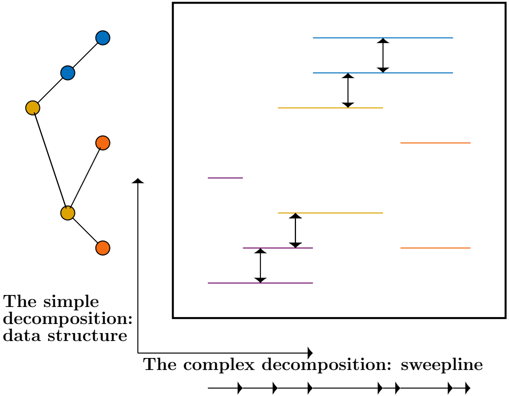
1. Batching: b binary search, $O(\log(n/b))$ depth, $O(b \log(n/b))$ work
2. Sweeping: $\sum_{k=1}^{b} O(\log(s_{(k-1)n/b} + n/b)$ work and depth
3. Refining: building the i-th prefix structure takes $O(\log s_{i-1})$ time. Total work $\sum_{k=1}^{n} O(\log(s_{k-1}))$, depth $\max_k \sum_{i=1}^{n/b} O(\log(s_{(k-1)n/b+i-1}))$.

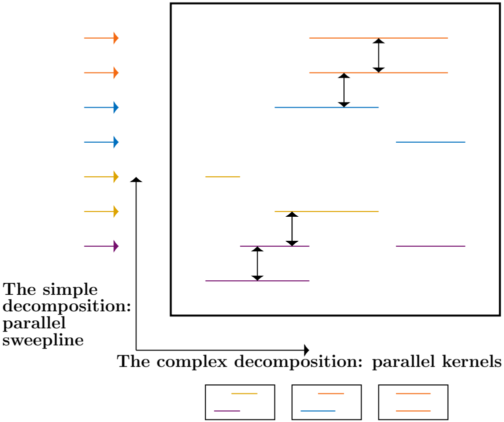
Note that $s_i = O(i)$.
The worse case: $O(n \log n)$ work and $O((b + n/b) \log n)$ depth.
When $b = \Theta(\sqrt{n})$, the depth is $O(\sqrt{n} \log n)$.

Insight: Designing Parallel Algorithms

- Decompose a problem by the ‘simple’ direction for parallelism, and leave the ‘complex’ work to each individual processor.
- The emphasis is different from the sequential version: we use sweepline to deal with the hard direction and maintain the easy direction for efficient query
- In the distance check case: horizontal is the hard direction (2 endpoints per segment, no total order)

Insight: Sequential vs Parallel





GPU Implementation

Implementation Details

- The sweepline framework is divide-and-conquer (GPU-friendly)
- dynamic algorithm selection: don’t invoke GPU if not necessary
- kernel granularity
  - tile-wise
  - polygon-wise
  - per prefix structure
  - per check
- Sorting?

Sorting on GPU

Two commonly used parallelizable sorting algorithms

- Merge sort
  - comparison-based
  - e.g., when you pass a *comparison function object* as an argument to `thrust::sort`
- Radix sort
  - non comparison-based
  - works for numeric data types (e.g., **`int`**) and default comparators

Copy-Sort-Permute 1/2

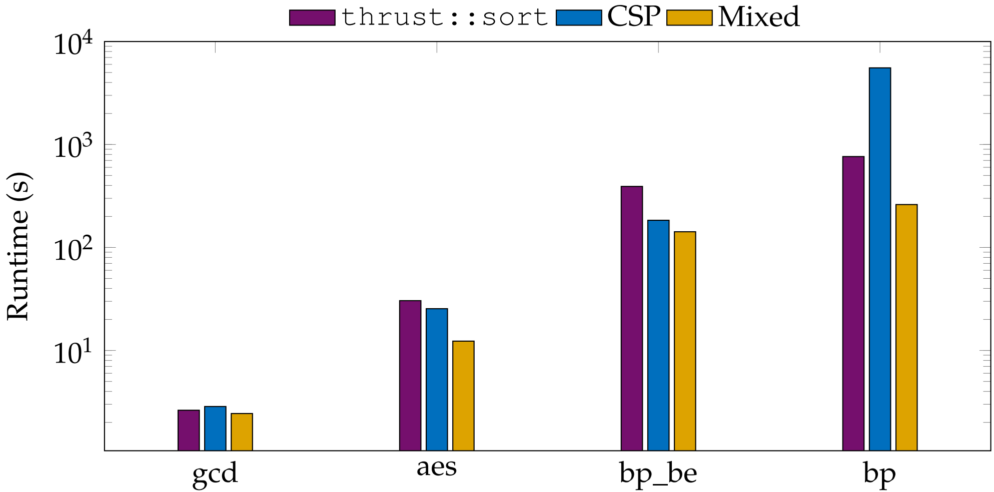
```
// Assume we want to sort array by S::key.
// n is the length of the array.
// effectively equivalent to thrust::sort(array, array+n);
template <typename S>
void sort_long_arrays(S *array, int n) {
  int *keys;      // the buffer for keys
  int *indices; // the buffer for indices
  S *tmp;         // the buffer for permutation

  // step 0: properly allocate the buffers
  cudaMallocManaged(...)...
```

Copy-Sort-Permute 2/2

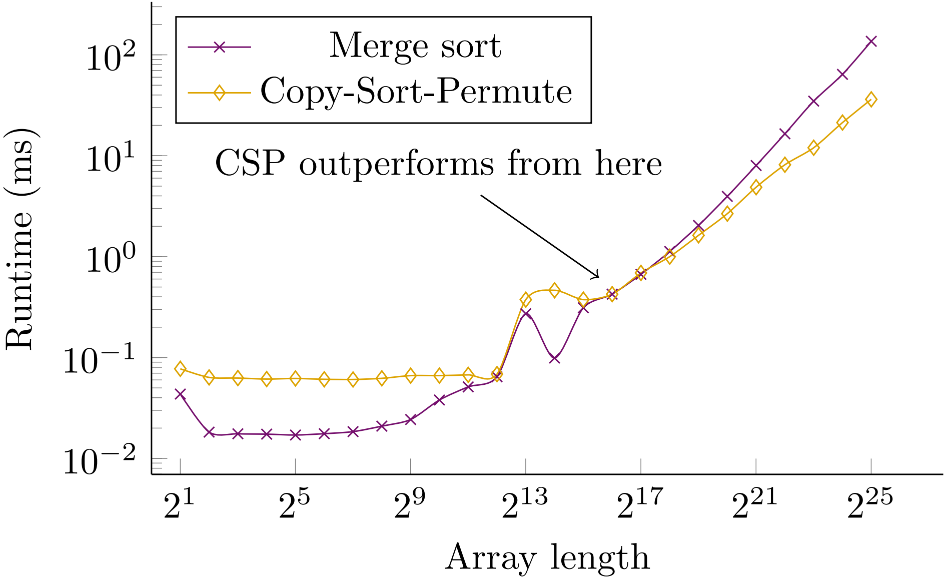
```
  // step 1: Copy
  for (int i = 0; i < n; ++i) {
    keys[i] = array[i].key;
    indices[i] = i;
  }
  // step 2: Sort
  thrust::sort_by_key(keys, keys+n, indices);
  // step 3: Permute
  thrust::copy_n(
      thrust::make_permutation_iterator(
          array, indices),
      n, tmp);
  thrust::copy_n(tmp, n, array);
}
```

Speedup by CSP



Runtime of enclosing check on `Metal 1` in log scale.

When to use CSP?



Experimental Results

Setup

- Implemented in C++ and CUDA
- Integrated into KLayout[1] (version 0.26.6)
  - Baseline: KLayout DRC Engine (8 threads)
- Test cases synthesized from OpenROAD[2]
- Environment:
  - Intel Xeon 2.90 GHz Linux machine with 128 GB RAM
  - One NVIDIA GeForce RTX 3090 GPU
  - NVCC 11.4, GNU GCC 10.3

[1] https://klayout.de
[2] https://github.com/The-OpenROAD-Project

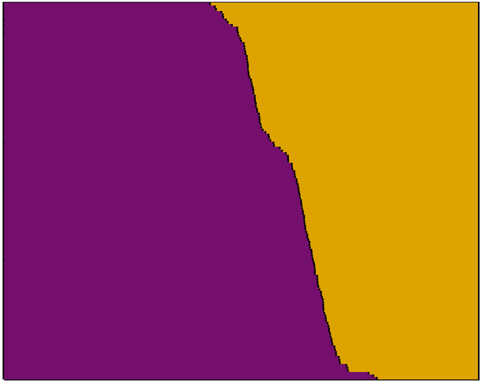
Stats and Width Check

Design	Layer	#Tiles	#Polygons	#Edges	#Edge/Polygon	Width Check Time (s) KLayout	Width Check Time (s) X-Check	Width Check Time (s) 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×

Enclosing Check and Space Check

Design	Layer	Enclosing Check			Space Check		
		KLayout	X-Check	Speedup	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	15470.4	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	66194.6	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	98370.4	235.3	418.06×	14019.7	233.4	60.07×
	Metal2	3958.7	276.6	14.41×	5164.4	65.9	78.37×
Average				61.36×			45.00×

Runtime Breakdown: Width Check

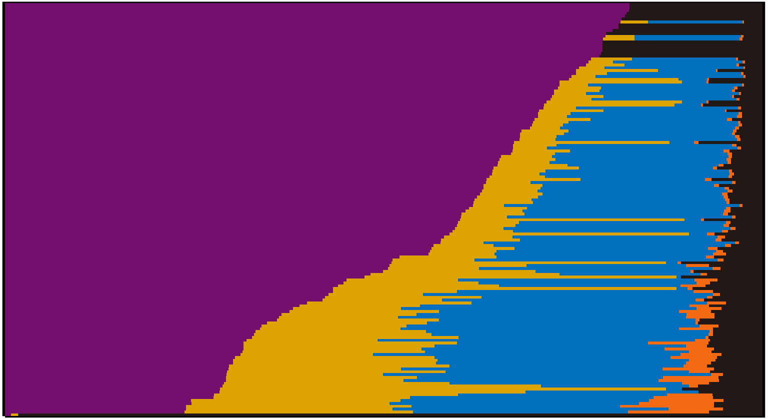


(a) KLayout

(b) X-Check

Each horizontal bar is for one thread. The purple and gold portions are for the *merge* and the check stages, respectively.

Runtime Breakdown: Enclosing Check



Each horizontal bar is for one thread. The purple portion is for *merge*, gold for *sort*, blue for *prefix build*, orange for *violation report*, and **black** for **the rest**, respectively.

Conclusion

- Parallel sweepline algorithm for DRC
- GPU implementation considerations
- Integration into an end-to-end flow
- Future work
    - Parallelize/Accelerate the merge stage
    - GPU infrastructure: associative data structures and thread-safe solution

THANK YOU!