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

Zhuolun He¹, Yuzhe Ma², Bei Yu¹

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

Sept. 14, 2022
Outline

1. Background and Motivation
2. Algorithm: Parallel Vertical Sweeping
3. GPU Implementation
4. Experimental Results
Background and Motivation
DRC: to ensure the layout does not violate geometric constraints

Typical rules: (a) width and spacing rules in a metal layer; (b) enclosing rule between a metal layer and a via layer.
• 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 (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 $\delta$. 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 $P$ by ascending $x$-coordinates
2. Initialize an empty BST $S$ (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 $S$ for $[y - \delta, y + \delta]$
      2. Report the corresponding segment pairs
      3. Insert $h$ to $S$
   2. Otherwise (i.e., right endpoint)
      1. Delete $h$ from $S$

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
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
   \[ s = (4, 9, 12, 18, 20, 25, 26, 27, 27) \]
Key idea: the prefix structure contains a set $S$ of segments that are below current segment within $\delta$ in $y$-direction

Remains to check if each pair of segments overlap in the $x$-direction
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 $O(\sqrt{n \log n})$. 
• 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)
The complex decomposition: sweepline
The simple decomposition: data structure

The complex decomposition: parallel kernels
The simple decomposition: parallel sweepline

Insight: Sequential vs Parallel
GPU Implementation
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?
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
// 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(...)

// 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);
Runtime of enclosing check on Metal 1 in log scale.

Speedup by CSP
When to use CSP?

CSP outperforms from here

<table>
<thead>
<tr>
<th>Array length</th>
<th>Runtime (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2^1</td>
<td>10^-2</td>
</tr>
<tr>
<td>2^5</td>
<td>10^-1</td>
</tr>
<tr>
<td>2^9</td>
<td>10^0</td>
</tr>
<tr>
<td>2^13</td>
<td>10^1</td>
</tr>
<tr>
<td>2^17</td>
<td>10^2</td>
</tr>
</tbody>
</table>

When to use CSP?
Experimental Results
• Implemented in C++ and CUDA
• Integrated into KLayout\textsuperscript{1} (version 0.26.6)
  • Baseline: KLayout DRC Engine (8 threads)
• Test cases synthesized from OpenROAD\textsuperscript{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

\textsuperscript{1}\url{https://klayout.de}
\textsuperscript{2}\url{https://github.com/The-OpenROAD-Project}
## Stats and Width Check

<table>
<thead>
<tr>
<th>Design</th>
<th>Layer</th>
<th>#Tiles</th>
<th>#Polygons</th>
<th>#Edges</th>
<th>#Edge/Polygon</th>
<th>KLayout Time (s)</th>
<th>X-Check Time (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>gcd</strong></td>
<td>Metal1</td>
<td>1</td>
<td>391</td>
<td>24440</td>
<td>62.5</td>
<td>&lt;0.1</td>
<td>0.1</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Metal2</td>
<td>1</td>
<td>1229</td>
<td>4916</td>
<td>4.0</td>
<td>&lt;0.1</td>
<td>&lt;0.1</td>
<td>-</td>
</tr>
<tr>
<td><strong>aes</strong></td>
<td>Metal1</td>
<td>16</td>
<td>17739</td>
<td>2059906</td>
<td>116.1</td>
<td>2.9</td>
<td>3.0</td>
<td>0.97×</td>
</tr>
<tr>
<td></td>
<td>Metal2</td>
<td>16</td>
<td>76007</td>
<td>304028</td>
<td>4.0</td>
<td>0.2</td>
<td>0.1</td>
<td>-</td>
</tr>
<tr>
<td><strong>bp_be</strong></td>
<td>Metal1</td>
<td>56</td>
<td>34747</td>
<td>27245522</td>
<td>784.1</td>
<td>21.9</td>
<td>19.3</td>
<td>1.13×</td>
</tr>
<tr>
<td></td>
<td>Metal2</td>
<td>56</td>
<td>393834</td>
<td>1575336</td>
<td>4.0</td>
<td>0.4</td>
<td>0.4</td>
<td>-</td>
</tr>
<tr>
<td><strong>bp</strong></td>
<td>Metal1</td>
<td>144</td>
<td>107706</td>
<td>52595418</td>
<td>488.3</td>
<td>38.9</td>
<td>33.0</td>
<td>1.18×</td>
</tr>
<tr>
<td></td>
<td>Metal2</td>
<td>144</td>
<td>833588</td>
<td>3334352</td>
<td>4.0</td>
<td>0.9</td>
<td>0.9</td>
<td>-</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.09×</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Design</td>
<td>Layer</td>
<td>Enclosing Check</td>
<td></td>
<td>Space Check</td>
<td></td>
<td>Speedup</td>
<td></td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>-------</td>
<td>-----------------------</td>
<td>----------</td>
<td>-------------------</td>
<td>----------</td>
<td>---------</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>KLayout</td>
<td>X-Check</td>
<td>Speedup</td>
<td>KLayout</td>
<td>X-Check</td>
<td>Speedup</td>
<td></td>
</tr>
<tr>
<td>gcd</td>
<td>Metal1</td>
<td>38.4</td>
<td>2.4</td>
<td>16.00×</td>
<td>12.6</td>
<td>2.4</td>
<td>5.25×</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Metal2</td>
<td>2.5</td>
<td>2.5</td>
<td>1.00×</td>
<td>6.4</td>
<td>2.4</td>
<td>2.67×</td>
<td></td>
</tr>
<tr>
<td>aes</td>
<td>Metal1</td>
<td>15470.4</td>
<td>12.3</td>
<td>1257.76×</td>
<td>4493.8</td>
<td>67.5</td>
<td>66.57×</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Metal2</td>
<td>2227.0</td>
<td>14.5</td>
<td>153.59×</td>
<td>2778.5</td>
<td>9.9</td>
<td>280.66×</td>
<td></td>
</tr>
<tr>
<td>bp_be</td>
<td>Metal1</td>
<td>66194.6</td>
<td>128.6</td>
<td>514.73×</td>
<td>6718.7</td>
<td>123.7</td>
<td>54.31×</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Metal2</td>
<td>3089.2</td>
<td>147.4</td>
<td>20.96×</td>
<td>4171.5</td>
<td>16.6</td>
<td>251.30×</td>
<td></td>
</tr>
<tr>
<td>bp</td>
<td>Metal1</td>
<td>98370.4</td>
<td>235.3</td>
<td>418.06×</td>
<td>14019.7</td>
<td>233.4</td>
<td>60.07×</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Metal2</td>
<td>3958.7</td>
<td>276.6</td>
<td>14.41×</td>
<td>5164.4</td>
<td>65.9</td>
<td>78.37×</td>
<td></td>
</tr>
<tr>
<td>Average</td>
<td></td>
<td>61.36×</td>
<td></td>
<td></td>
<td>45.00×</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Runtime Breakdown: Width Check

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