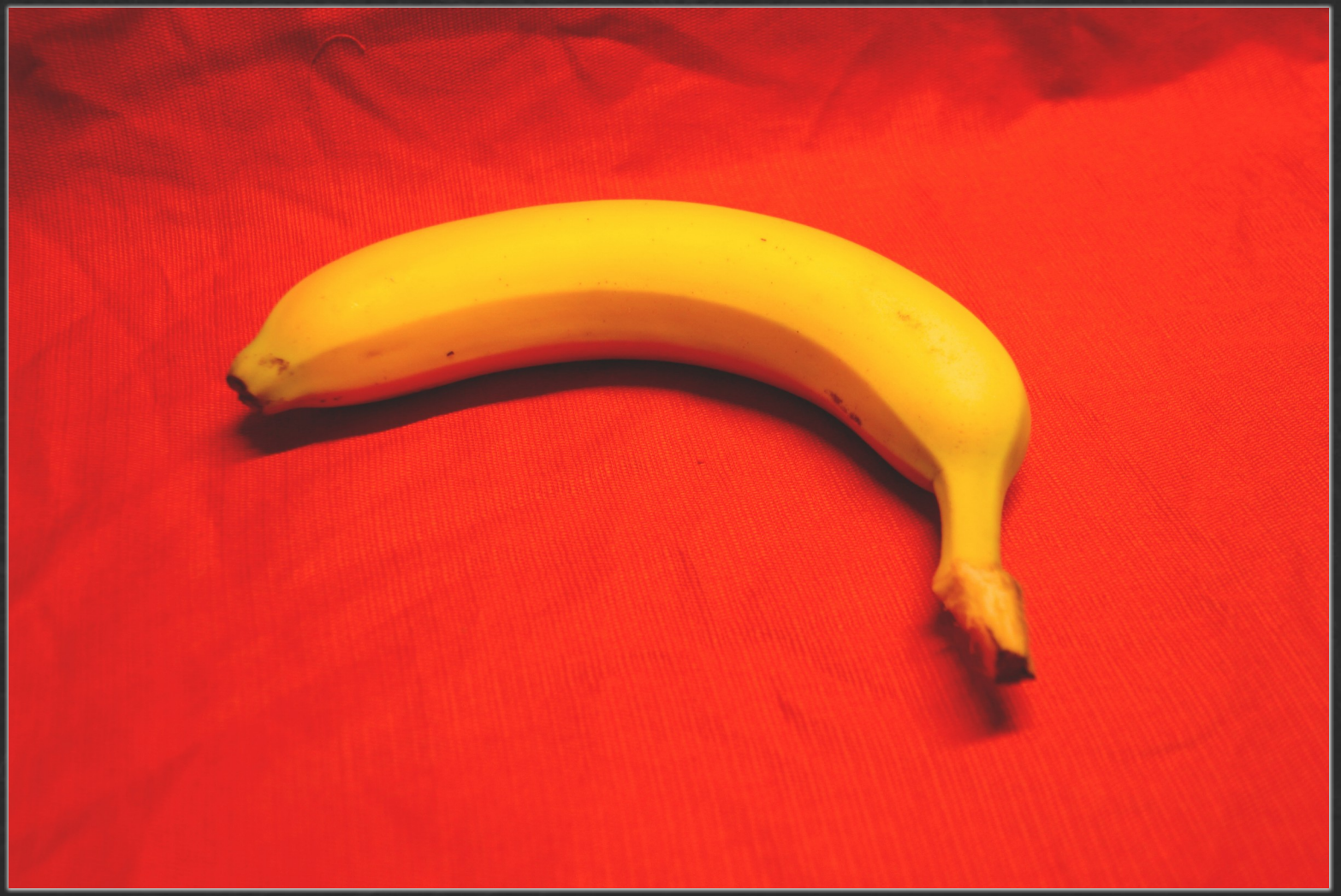
DRC/LVS Development Best Practices

Learning from GF180 PDK optimization

Matthias Köfferlein, https://www.klayout.org

This is **not** a Banana!



The GF180 DRC/LVS

- Project link (efabless fork)

  https://github.com/efabless/globalfoundries-pdk-libs-gf180mcu_fd_pv

- Highlights
  - Nice Python wrapper
  - Large test suite
  - Modular
  - Clean structure
  - References to design manual
- Troubles
  - Performance issue (>10h runtime, >40G memory for medium size layout with 410k stdcells)

Performance Killers

- KLayout bug (garbage collector disabled) **Fixed** ✔
  - Pile-up of memory for intermediate results
- Use of flat mode **Fixed** ✔
  - little trust in the other modes?
- Inefficient implementation of certain rules

- After optimization (large test case)
  - speed 10h → 1h
  - memory 40G → <4G
  - runs on single CPU and consumer hardware

Debugging Techniques

Recent features make debugging easier

`profile`

Used at the beginning of a script will print the commands by CPU time and process memory delta

```
Operations by execution time

Operation                                  # calls  Time (s)  Memory (k)
"enclosing" in: sky130A_mr.drc:395         1        72.930    0
"-" in: sky130A_mr.drc:397                 1        70.440    0
"enclosing" in: sky130A_mr.drc:396         1        62.270    0
"enclosing" in: sky130A_mr.drc:388         1        41.010    0
"&" in: sky130A_mr.drc:290                 1        38.710    32784
"space" in: sky130A_mr.drc:374             1        26.070    32784
"enclosing" in: sky130A_mr.drc:449         1        22.150    0
"-" in: sky130A_mr.drc:419                 1        19.950    32784
"enclosing" in: sky130A_mr.drc:384         1        16.750    0
"width" in: sky130A_mr.drc:368             1        16.380    0
"enclosing" in: sky130A_mr.drc:421         1        15.030    0
"enclosing" in: sky130A_mr.drc:418         1        14.380    0
"-" in: sky130A_mr.drc:289                 1        14.310    -131136
"space" in: sky130A_mr.drc:435             1        13.240    0
"width" in: sky130A_mr.drc:428             1        13.190    0
"interacting" in: sky130A_mr.drc:299       2        12.760    0
"interacting" in: sky130A_mr.drc:290       1        12.570    5892
"width" in: sky130A_mr.drc:265             1        11.260    0
"without_length" in: sky130A_mr.drc:294    1        10.070    94792
```

Memory returned to system by garbage collector

`new_target`

Allows sending intermediate results to a separate layout file for easy inspection

```
1
2 report("L1 over L2 overlap")
3
4 l1 = input(1, 0)
5 l2 = input(2, 0)
6
7 debug = new_target("debug.gds")
8
9 l1and2 = l1 & l2
10
11 l1and2.output(debug, 1, 0)
12
13 l1and2.width(1.0.um).output("l1 over l2 < 1µm")
14
```

Choice of Modes

`flat` (default)	`tiled`	`deep`
Simple, predictable, single CPU, vanilla implementation	Operations work on tiles	Hierarchical processing where possible (local computation done once per cell)
Memory proportional to # objects	Parallelization along tiles, good scaling	Can be very fast, but also slow (skillful use reqd)
Only for small designs or quick checks	Heap allocation for single tiles only	Results / intermediate layers are hierarchical → small memory footprint possible
	Results / intermediate layers are flat → large memory footprint possible	Scales with “cores $^{0.5}$”
	Range-limited (border specification needed)	**Preferred solution for big hierarchical layouts**
	Useful for flat layouts	

Deep Mode in a Nutshell

A OP **B**

*) OP = “local”

```
compute(A, B, OP, dist):

  for subject in shapes of A:
    intruders = shapes of B with distance to subject < dist
    results = OP.compute(subject, intruders)
    store results
```

“Visitor pattern”

Hierarchical treatment

- Compute cell neighborhoods (“contexts”)
- Collect intruders per context (→ minimum set of configurations)
- For the results, keep common core inside cell,
  propagate specific results to parent cells

Some

Deep Mode Best Practices

- Watch for hierarchy degradation
  - Results my be propagated, destroying hierarchy over time

```
"-" in: sky130A_mr.drc:294
    Polygons (raw): 682248 (flat)  329 (hierarchical)
    Elapsed: 0.020s  Memory: 4391.00M
```

- Complexity determined by first operand
  - Less shapes, less work
  - The more hierarchical, the better
  - First operand is able to “pull” B shapes down in hierarchy
- Beware of pre-merge
  - Not all operations are “local” and need pre-merge - e.g. “interact”
  - Pre-merge will form large polygons potentially higher up in the hierarchy → spoils hierarchical performance

For details see: https://www.klayout.de/drc_function_internals.html#drc_function_details

Klayout is **not** Calibre!

Klayout is **not** Calibre!

- Immediate execution vs. operation graph
  - Layer == Variable, Value == Layer Geometry
  - Memory allocation == variable lifetime → use “forget” or reset variable
  - Intermediate results allocate memory too (will be cleaned up by GC)

```
d = a - (b & c)
```

Intermediate result - avoid duplication of expressions

  - No optimization of dead execution branches

```
c = empty & a.interacting(b)
```

Computed even though not needed

  - No selection of input layers based on what is needed
  - No parallelization
  - **Pro:** allows loops, conditionals and direct per-shape manipulations
- No hierarchy manipulation
  - Except for variant formation for non-isotropic transformations and grid snap operations

Pitfalls we have seen

- “drc” function is more generic, but not better than simple equivalents

```
a.drc(space < 0.2.um)
```

```
a.space(0.2.um)
```

Same result, but performance is better with “space”

- Edge “width” != polygon “width”
  - Edge “width” only refers to relative orientation of the edges, but treats edges separately (→ potential long-distance interactions)
  - Polygon “width” is a single-polygon operation on pre-merged polygons

```
a.edges.width(0.5.um)
```

```
a.width(0.5)
```

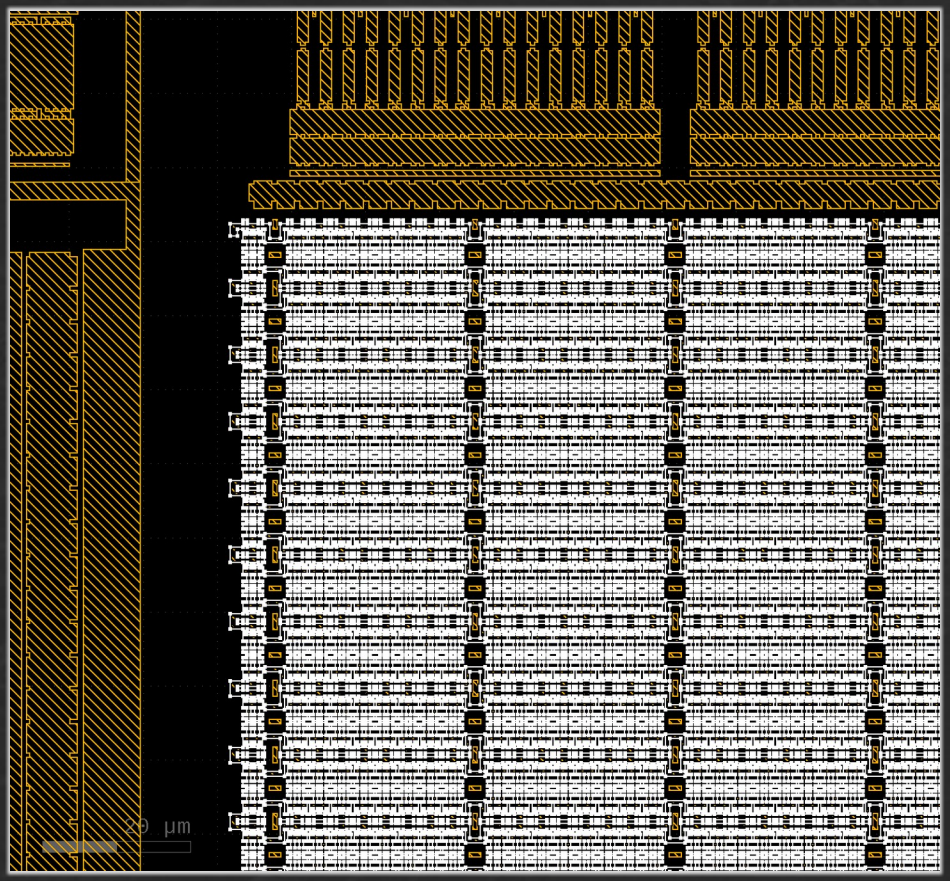
Similar results, but left side is better with large clusters of polygons while right side is better with large distances

- “+” (join) may be better than “|” (or)
  - “+” simply collects the shapes, “|” merges the shapes → this may give large polygons high up in the hierarchy and eats CPU time
  - For “local” operations, fragmented input is better → use “+”

“+” (join) vs. “|” (or)

`poly.or(comp)`

Gives a single giant polygon over memory area



`poly + comp` ✔

Leaves the original polygons in the hierarchy

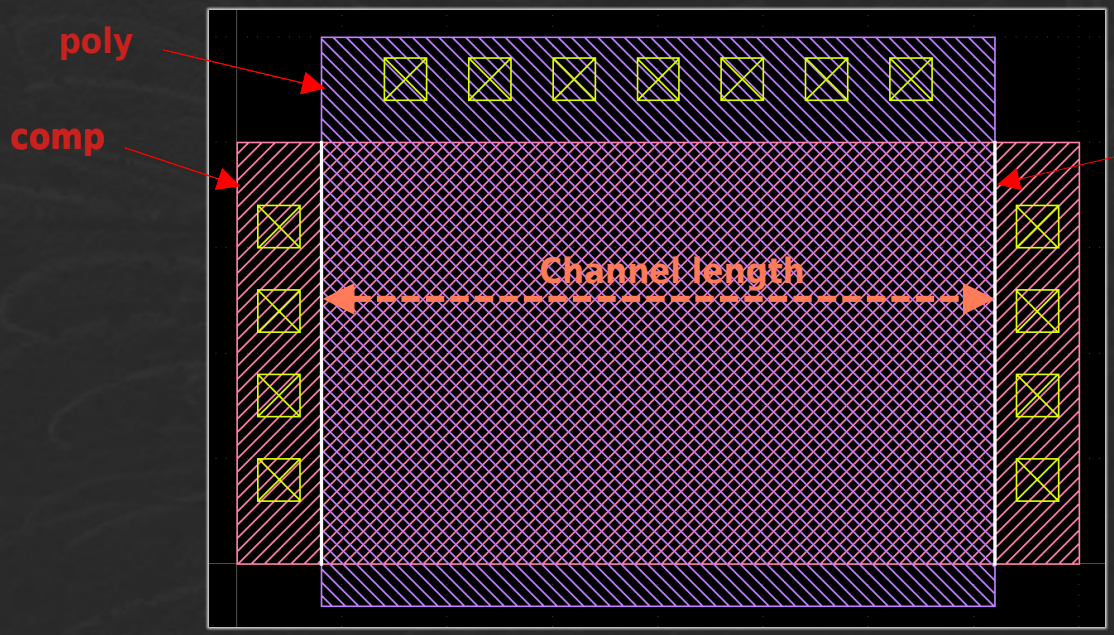
Executes much faster on operations not doing pre-merge

Optimization Example I

Rule: Max transistor channel length <= 20 µm

Initial implementation (concept):

```
channel_edges = poly.edges & comp
channel_not_too_wide = channel_edges.width(20.001.um)
error = channel_edges -
  channel_edges.interacting(channel_not_too_wide.edges)
```



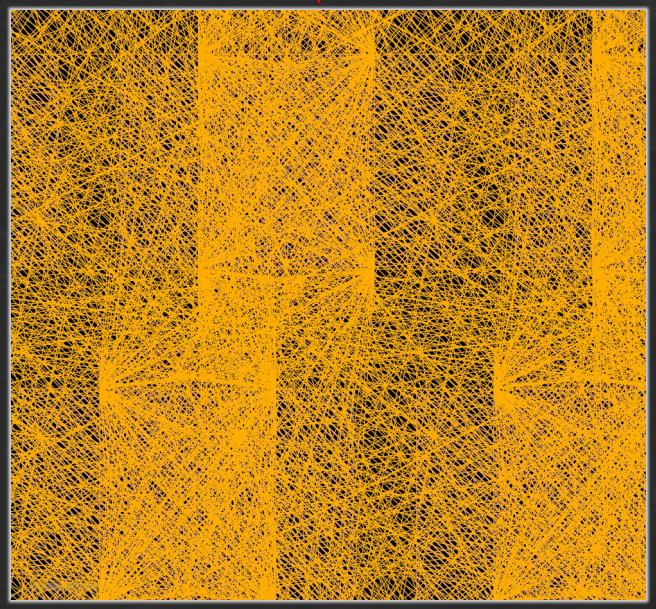
Observation

slow execution on standard logic layouts

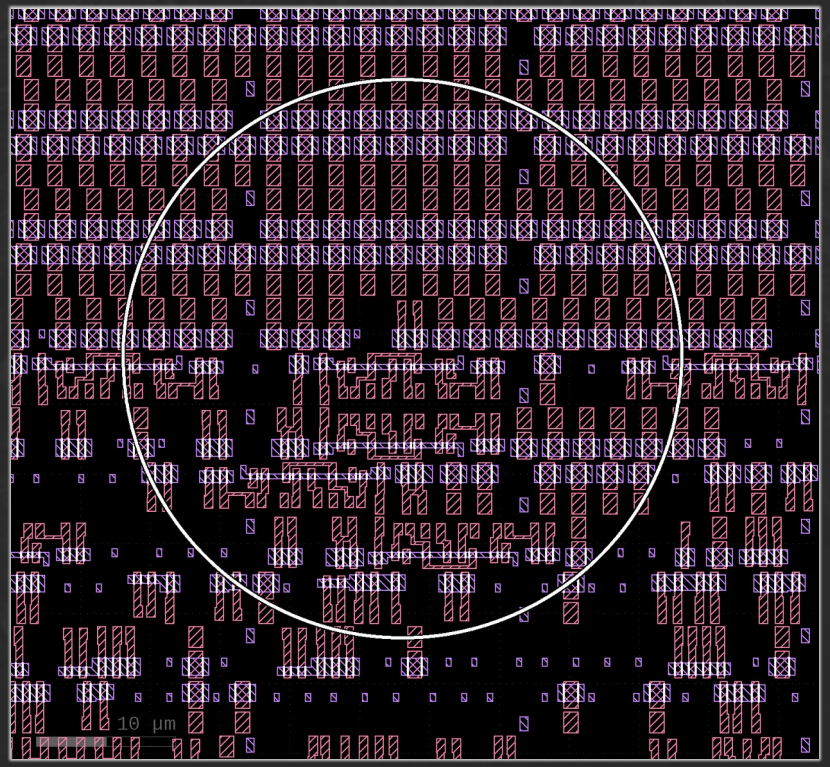
Even worse in deep mode

Analysis

```
channel_edges = poly.edges & comp
channel_not_too_wide = channel_edges.width(20.001.um)
error = channel_edges -
  channel_edges.interacting(channel_not_too_wide.edges)
```



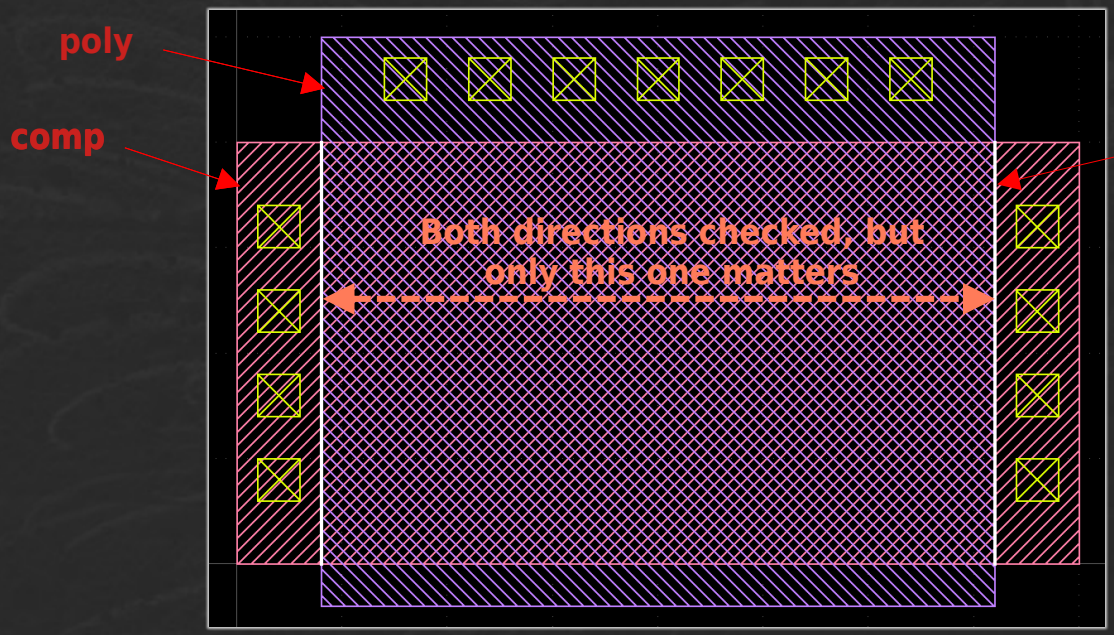
Explanation: edge “width” captures many interactions due to the long range



Optimized Version

Rewriting to polygon width check → range is limited to polygon area

```
channel_edges = poly.edges & comp
gate = comp & poly
gate_not_too_wide = gate.width(20.um + 1.dbu,
                               projection)
error = channel_edges - gate_not_too_wide.edges
```



Effect ✔

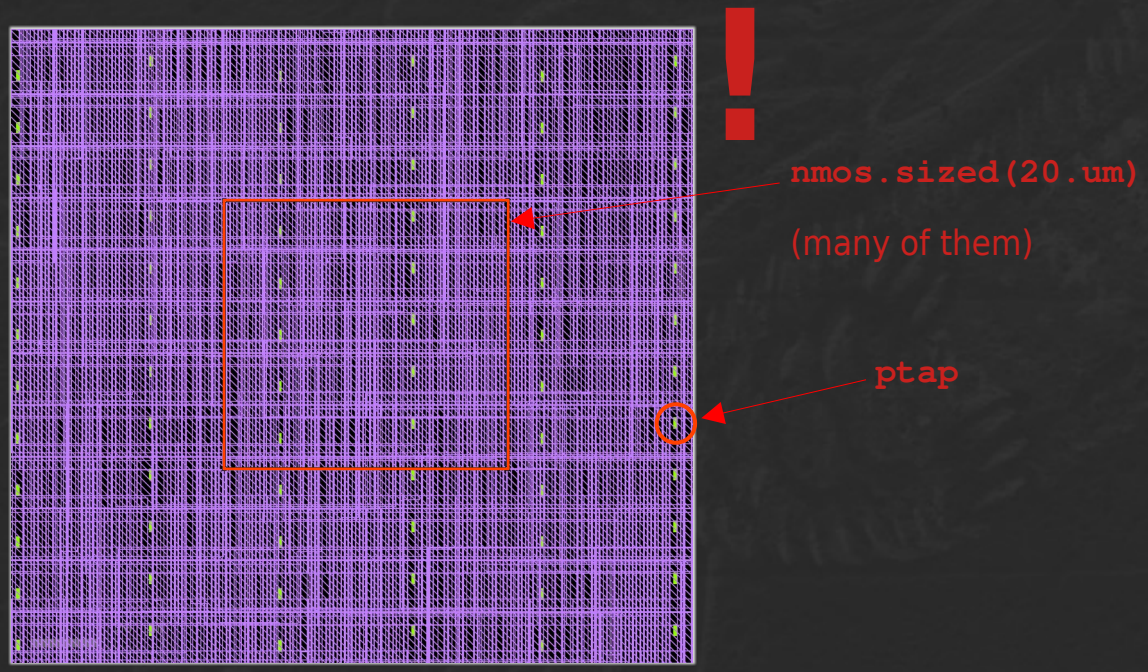
Execution time drops from 50s (medium size sample) to basically nothing

Optimization Example II

Rule: NMOS distance to p tap <= 20µm

Initial implementation (concept):

```
nmos = ncomp.outside(nwell)
ptap = pcomp.outside(nwell)
error = ptap.not_interacting(nmos.sized(20.um))
```



Observation

slow execution on standard logic layouts

Even worse in deep mode

Optimized Version

Turning around the check optimizes it

Explanation

- ptap has less shapes than nmos
- ptap is localized → pre-merge of “sized” does not spoil the hierachy and is quick.

Effect: “nmos.not_interacting(...)” has more primary shapes, but has to deal with fewer intruder shapes

```
nmos = ncomp.outside(nwell)
ptap = pcomp.outside(nwell)
error = nmos.not_interacting(ptap.sized(20.um))
```

Effect

Execution time
drops by a factor 10 ✔

Wrap-up

- Prefer deep mode
- Keep in mind the basic concepts of deep mode
  - First argument should have low complexity
  - Beware of large regions formed by pre-merge
  - Avoid hierarchy degradation
- Use profiling, focus on the greedy ones
- Look at the intermediate results
- Rethink your rule implementation & try alternatives
- LVS: needs hierarchical device recognition layers for schematic / layout correspondence
  - Avoid hierarchy degradation (specifically pre-merge driven)

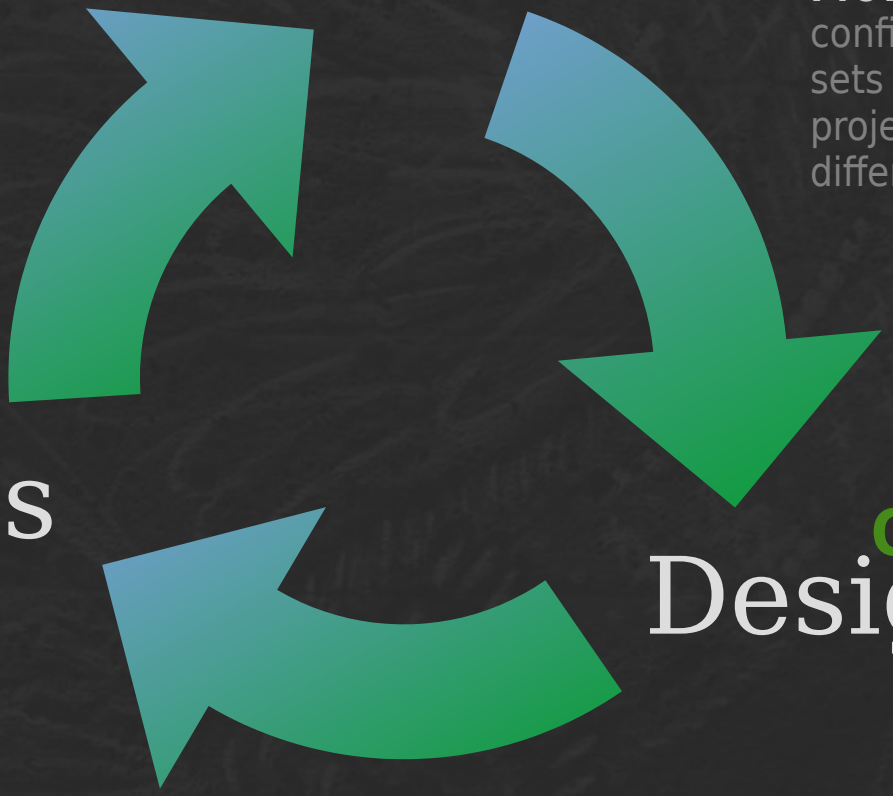
Homework

- What are the input / output formats in different tools?

Input / Output	KLayout	Others
Layout	GDS2 / OASIS etc.	GDS2 / OASIS etc.
DRC / LVS decks	Ruby language, tool specific, but follows conventions	Proprietary, copyright protected
Error DB	Tool specific, documented	Proprietary
LVS database	Tool specific, (documented)	Proprietary

- Need help from community to enable some features?
  - YES preferably in the form of test cases, benchmarks, user stories and problem statements
  - YES in form of scripted prototypes
  - (under certain conditions) C++ core features
- Is a common (open source) database a solution to some of the (open) questions?
  - In parts it is where no open standards exists
  - But after all, a “silver bullet” does not exist – IMHO we get more value if we focus on making best use of what we have and seamless integration

Vision: The Open Source Growth Cycle



Enable open and worldwide collaboration on development of the PDK, co-development of tools and PDK features

Provide standardized configurations and tool sets for a large number of projects and users across different cultures

Provide test cases, regression tests, benchmarks, use cases, user stories, feedback & defect reports

Thank you for Listening!