Exploring Parallelism Opportunities in KLayout (project report, group 22)

Zeren Chen
zerenat.owlfox<dot>org
Taiwan

ChatGpt4
Open AI
USA

MH Yan
Taiwan

ABSTRACT

This project investigates the application of parallelism to enhance performance in KLayout's Design Rule Checking (DRC) processes. Initially, we aimed to bolster single DRC operation parallelism using OpenMP. However, complexities in the C++ code base and race conditions posed challenges. We then pivoted to Taskflow, utilizing its task-level parallelism capabilities, which demonstrated encouraging performance improvements in our preliminary experiments. The project's code and experiment scripts are available at https://github.com/owlfox/pps23_final_project.

KEYWORDS

KLayout, DRC, Parallel Programming, OpenMP, Taskflow, Open-Source, Work-In-Progress

1 INTRODUCTION

With the escalating complexity of contemporary integrated circuit designs, the need for expediting DRC processes has become imperative. In light of this, we selected KLayout—a GPL open-source C++ layout viewer/editor/analyzer first released in April 2006 and maintained by Matthias **K**oefferlein—for our project. Our objective was to ascertain the feasibility of applying parallelism techniques learned in classrooms to a real-world C++ code base.

2 PROPOSED SOLUTION

Initially, we intended to inject parallelism into KLayout's DRC scripts using OpenMP, focusing primarily on the DRC code that computes derived layer shapes from two or more input layers. For instance:

Listing 1: DRC code used for benchmarking in KLayout

```
m1 = polygons(68, 20)
via = polygons(68, 44)

gate = diff.and(poly)
```

Our goal was to pinpoint the slowest part of the 'and' operation code, using real-world design input—caravel.gds.gz from the SkyWater 130nm openPDK. Despite the promise of this approach, we encountered challenges in ensuring data safety when adding OpenMP pragma into the C++ function in listing 2. This code segment is where the "and" operation spent most of its runtime based on our perl and FlameGraph[4] profiling result. You can run ./drc.sh to reproduce the experiment for the original benchmarking setup. Consequently, we turned to Taskflow—a parallel and heterogeneous programming framework. Taskflow provided a more effective solution for parallelizing DRC scripts.

Listing 2: dbHierProcessor.cc

```
template <class TS, class TI, class TR>
    void local_processor<TS, TI, TR>::
    compute_local_cell (const db::local_
    processor_contexts<TS, TI, TR> &
    contexts, db::Cell *subject_cell,
    const db::Cell *intruder_cell, const
    local_operation<TS, TI, TR> *op, const
     typename local_processor_cell_
    contexts<TS, TI, TR>::context_key_type
     &intruders, std::vector<std::
    unordered_set<TR> > &result) const {/*
    ...*/}
```

3 EXPERIMENTAL METHODOLOGY

Our experimental methodology involved implementing Taskflow for task-level parallelism in DRC scripts and comparing the execution times of these parallelized versions with their serial counterparts. The testbed for our experiments was a multicore system with the following specifications:

- CPU: 12th Gen Intel(R) Core(TM) i7-12700K
- Memory: 32 GB DDR4
- GPU: NVIDIA GeForce RTX 3050 (Driver Version: 530.41.03, CUDA Version: 12.1)
- OS: proxmox-ve 7.3-1 (running kernel: 5.15.35-1-pve)

Unlike our initial approach with OpenMP, we decided to apply Taskflow to a specific digital design cell. This strategic shift allowed us to reduce the Turn-Around-Time (TAT) significantly, while simultaneously confirming the feasibility of using Taskflow. The sample DRC C++ code we used is illustrated below, with both its serial and Taskflow versions:

Listing 3: DRC operation using both Serial and Taskflow approaches

```
// Serial version
db::Region l1 = all_m1 & all_poly;
db::Region l2 = all_m1 & all_diff;
db::Region l3 = all_poly & all_via;
db::Region l4 = all_m1 & all_via;

// TaskFlow version
tf::Executor executor;
tf::Taskflow taskflow("simple");
auto [A, B, C, D] = taskflow.emplace(
  [&l1, &all_m1, &all_poly]() { l1 = all_
      m1 & all_poly; std::cout << "TaskA\n
      "; },
  [&l2, &all_m1, &all_diff]() { l2 = all_
      m1 & all_diff; std::cout << "TaskB\n
      "; },
  [&l3, &all_via, &all_poly]() { l3 = all_
      poly & all_via;  std::cout << "TaskC
      \n"; },
  [&l4, &all_m1, &all_via]() { l4 = all_m1
       & all_via; std::cout << "TaskD\n";
      }
);
executor.run(taskflow).wait();
```

The Taskflow version was run on the executor, and the tasks were scheduled and executed based on their dependencies.

4 EXPERIMENTAL RESULTS

Our initial efforts to parallelize using OpenMP were met with considerable challenges, predominantly due to side effects and the complexity of the code base, hindering effective parallelization of the ’compute_local_cell’ function.

In the absence of task-level parallelism, the process recorded a grand total time of 0.517 seconds (0.49 seconds user, 0.03 seconds sys). Transitioning to Taskflow led to considerable performance enhancements. The same process, when executed with Taskflow, required a grand total time of only 0.111 seconds (0.19 seconds user, 0.01 seconds sys), demonstrating Taskflow’s proficiency in managing task-level parallelism.

The results evidence a significant speedup in executing DRC scripts using Taskflow. Furthermore, we observed superior scalability with an increasing number of processor cores, suggesting that our parallelization strategy has the potential for greater efficiencies with increased hardware resources.

5 RELATED WORK

During our project, it came to light that KLayout already employs parallelism techniques, such as Pthreads and tiling with the scanline algorithm, for DRC operations. This was not realized during the initial project proposal stage.

6 CONCLUSIONS AND FUTURE WORK

Our exploration into the parallelization of KLayout’s DRC processes yielded enlightening results. The challenges encountered with OpenMP underscored the critical role of selecting the right tool for parallel computing. Our subsequent adoption of TaskFlow demonstrated significant performance improvements, attesting to its potential for task-level parallelism management. However, the observed user time discrepancy between the serial and TaskFlow versions calls for a more comprehensive understanding of the underlying mechanisms.

Future work will involve investigating the user time discrepancy between the two versions. Potential factors could include differences in thread scheduling, parallel overheads, or other aspects unique to the KLayout code base and TaskFlow. We also aim to delve further into the KLayout code base and identify additional opportunities for optimizing our parallelization strategy. Ultimately, the goal is to improve the robustness of the project and ensure consistent speed-ups across various integrated circuit designs.

This work lays the foundation for further research on parallelizing integrated circuit design and testing processes, contributing to the ever-increasing demand for efficient design rule checking in the semiconductor industry.

REFERENCES

[1] KLayout Documentation, ”KLayout User Manual,” [Online]. Available: https://www.klayout.de/doc/index.html [Accessed: 04-April-2023] KLayout Repository, ”KLayout Source Code,” [Online]. Available: https://github.com/KLayout/klayout [Accessed: 04-April-2023].
[2] Taskflow, ”Taskflow: A General-purpose Parallel and Heterogeneous Task Programming System,” [Online]. Available: https://taskflow.github.io/ [Accessed: 04-April-2023].
[3] SkyWater Open Source PDK, ”SkyWater Open Source PDK Documentation,” [Online]. Available: https://skywater-pdk.readthedocs.io/en/main/ [Accessed: 05-June-2023].
[4] Brendan Gregg, ”FlameGraph,” [Online]. Available: https://github.com/brendangregg/FlameGraph [Accessed: 05-June-2023].