

CS516000 FPGA Architecture & CAD

Assignment – Congestion Driven FPGA Placement

賈俊佑 studentID:114062645

1.Introduction

Implements a congestion-driven placer for an island-style FPGA. Given an $R \times C$ grid of logic block sites, a set of movable logic blocks, and fixed I/O pins, the objective is to produce a legal placement that minimizes both Half-Perimeter Wire Length (HPWL) and routing congestion.

2.Objective

The primary goal of this assignment is to develop a placer that satisfies the following three key requirements:

- 1) Legal Placement
- 2) Minimize Congestion
- 3) Minimize HPWL

3. Methodology

This work implements a simulated-annealing (SA) based congestion-driven placer for island-style FPGAs. The overall flow follows a VPR-like SA framework [2], integrates a congestion cost inspired by congestion-driven placement for FPGAs [1], and adopts a region-based move operator similar in spirit to recent SA improvements [3].

3.1 Initial Placement

The placer starts from a simple random legal initial placement. All movable logic blocks are collected into a list, and all available grid sites in the FPGA array are enumerated. The list of sites is randomly shuffled, and each logic block is assigned to a unique site in this order, while fixed I/O blocks remain at their predefined positions.

3.2 Simulated Annealing Framework

Starting from the initial placement, the tool applies a standard simulated annealing (SA) framework [2][3]. At the beginning, the temperature is initialized so that a high fraction of uphill moves is accepted, enabling broad exploration of the search space. For each temperature, an adaptively adjusted number of moves is attempted. These moves are generated based on a dynamic range limiter and a probabilistic region selection strategy; each move is evaluated using the cost function in Section 3.3 and is accepted if it lowers the cost, otherwise it is accepted only with a probability that becomes smaller for worse moves and at lower temperatures.

After each temperature step, the temperature is reduced according to a geometric-like cooling schedule until either a minimum temperature is reached or the time limit expires. When the annealing process terminates, the placement obtained at the final temperature is taken as the output solution. Detailed parameter settings (initial temperature estimation, cooling rate, and the number of moves per temperature) are given later in the Implementation Details section.

3.3 Cost function

The simulated annealing engine in this work minimizes a single scalar cost function that jointly captures wirelength and routing congestion:

$$Cost = Total\ HPWL \times Congestion\ Coefficient.$$

Total HPWL: For each net, a bounding box is computed from the positions of its pins, and the half-perimeter wirelength (HPWL) is accumulated over all nets. This is the standard wirelength proxy widely used in FPGA placement and in VPR-style frameworks [2].

Congestion Coefficient: Following the nets-overlap idea in [1], every net increases the usage of all tiles inside its bounding box, forming a 2D usage map. Simple statistics of this map are then collapsed into a scalar congestion coefficient that grows when routing demand becomes uneven and strongly concentrated in a few regions [1][2].

Multiplying HPWL by this coefficient encourages solutions that are not only short in total wirelength but also more evenly distributed in routing demand.

In addition, the implementation also evaluates the k-power variant proposed in [1]:

$$Cost = Total\ HPWL \times (Congestion\ Coefficient)^k \quad k \in \{1,2,3\}$$

where fixed k values in $\{1,2,3\}$ are tested, and an adaptive scheme is implemented in which k increases with the size of highly congested areas and decreases as these areas shrink, following the idea in [1]. However, across the benchmark circuits this adaptive-k formulation does not produce consistent improvements: in many cases the wirelength slightly worsens and the annealing becomes less stable. Therefore, all reported experiments adopt the simpler and more robust form with fixed $k = 1$.

3.4 Region-Based Move Operator and Incremental Update

The move generator in this work employs a hybrid strategy that alternates between directed region-based moves and standard local random moves. In each simulated annealing step, a single movable logic block is selected, and a decision is made probabilistically whether to perform a Region Move or a Local Move:

- Region Move: If selected, a candidate region is calculated based on the connectivity of the block (as detailed below) to guide the block toward an "ideal" location.
- Local Move: Otherwise, a search window is centered around the block's current position to perform a local neighborhood exploration.

Once the search region is defined, a target site is sampled uniformly within it. The block is then either moved into an empty site or swapped with the block currently occupying that site. Only nets incident to the moved/swapped blocks are re-evaluated, and their contributions to the global cost are updated incrementally [1][2].

Inspired by the optimal-region concept in [3], an Optimal Region Move is first implemented, where the region is defined from the median of the bounding-box edges of all nets connected to the selected block. In theory, any position within this “optimal region” should not increase the HPWL for those nets [3]. However, experimental results show that this strategy does not consistently outperform a simpler averaging-based region.

Therefore, the final placer adopts a new averaging-based scheme referred to as the Average-Center Region (ACR) move. In the ACR move, the center of each incident net’s bounding box is computed, and these centers are averaged to obtain an “ideal” target point for the block. A rectangular search window around this average center is then used as the region from which the new site is sampled.

All experiments are run under the same SA configuration; only the region-move strategy (Optimal Region vs Average-Center Region) is changed.

	alu4		apex4		clma_2		Elliptic		pdc	
Optimal region	2466	1.069	3783	1.040	25322	1.032	5031	1.043	14674	1.049
Average-Center Region	2591	1.058	3851	1.041	25784	1.029	5111	1.042	15664	1.044

4. Implementation Details

4.1 Global Limits and Stopping Criteria

The placer is constrained by two global stopping conditions:

Time limit: the total runtime is limited to 220 seconds

$$\text{time_limit_seconds} = 220.0.$$

At each temperature round, the elapsed time is checked; if it exceeds this limit, the SA loop is terminated immediately.

Minimum temperature: the outer SA loop continues as long as

$$T > \text{min_temperature},$$

where

$$\text{min_temperature} = 1 \times 10^{-5}.$$

Once the temperature drops below this threshold, further cooling is considered unproductive and the algorithm stops.

4.2 Initial Temperature Estimation

The initial temperature is estimated automatically from the current placement by sampling a fixed number of random moves:

A total of 1000 random swap moves is performed between uniformly chosen logic blocks and random target sites on the chip. For each tentative move, the cost difference is computed using the cost function in Section 3.3. Only uphill moves with $\Delta C > 0$ are accumulated:

$$\Delta \bar{C}_+ = \frac{1}{N_+} \sum_{\Delta C > 0} \Delta C,$$

where N_+ is the number of uphill samples.

The target initial acceptance probability for uphill moves is set to

$$p_0 = 0.9,$$

and solve for T_0 from

$$p_0 = \exp \left(-\frac{\Delta \bar{C}_+}{T_0} \right).$$

This gives

$$T_0 = -\frac{\Delta \bar{C}_+}{\ln(p_0)}.$$

4.3 Moves per Temperature

The number of attempted moves per temperature, denoted as `moves_per_temperature`, is also controlled adaptively:

Initial value:

$$\text{initial_moves_per_temp} = 60000.$$

Hard bounds:

$$\text{min_moves_per_temp} = 40000.$$

$$\text{max_moves_per_temp} = 80000.$$

Scaling factors:

$$\text{moves_scale_up} = 1.2.$$

$$\text{moves_scale_down} = 0.8.$$

At the end of each temperature round, the acceptance rate is computed as

$$\text{Acc} = \frac{\text{accepted_moves}}{\text{moves_per_temperature}}$$

The move budget is then updated as follows. If the acceptance rate is too high or too low

$$\text{Acc} > 0.85 \text{ or } \text{Acc} < 0.05$$

the move count is decreased:

$$\text{moves_per_temperature} \leftarrow \text{moves_per_temperature} \times \text{moves_scale_down}.$$

Otherwise, when the acceptance rate is in a more productive middle range, the move count is increased:

$$\text{moves_per_temperature} \leftarrow \text{moves_per_temperature} \times \text{moves_scale_up}.$$

Additionally, when 90% of the time limit is consumed, the number of moves is forced to `min_moves_per_temp` to accelerate convergence before the hard timeout.

4.4 Region Probability Schedule

The region-based move operator (Section 3.4) is controlled by a probability `region_prob`, which determines how often using a directed region move (Optimal Region or Average-Center Region) instead of a purely local move:

Initialization and bounds:

`initial_region_prob=0.5.`

`min_region_prob=0.5.`

`max_region_prob=0.9.`

Step size:

`region_prob_step_up = region_prob_step_down=0.02.`

At the end of each temperature round, the acceptance rate `Acc` is again used to adjust `region_prob`. If the annealer is either too random or too frozen

$Acc > 0.85$ or $Acc < 0.10$

the use of region moves is slightly decreased:

`region_prob←region_prob - region_prob_step_down.`

Otherwise, when the acceptance rate is moderate, the use of region moves is slightly increased:

`region_prob←region_prob + region_prob_step_up.`

4.5 Temperature Cooling Schedule

The cooling rate is also adapted based on the observed acceptance rate. At the end of each round, a multiplicative factor α is chosen and the temperature is updated as

$$T \leftarrow \alpha \cdot T.$$

The factor α is selected according to:

$$\alpha = \begin{cases} 0.5, & \text{if } \text{Acc} > 0.96, \\ 0.85, & \text{if } 0.80 < \text{Acc} \leq 0.96, \\ 0.9, & \text{if } 0.15 < \text{Acc} \leq 0.80, \\ 0.85, & \text{if } \text{Acc} \leq 0.15. \end{cases}$$

4.6 Range Limiter Schedule

The range limiter controls the typical spatial extent of each move and is used to set the search radius for both local and region-based moves. It is initialized and bounded as:

Initial value:

$$\text{initial_range_limiter} = 1.0.$$

Bounds:

$$\text{min_range_limiter} = 0.01.$$

$$\text{max_range_limiter} = 1.0.$$

At each temperature round, the range limiter is updated as a function of the current temperature T and the initial temperature T_0 :

Compute a normalized scale factor

$$s = \max \left(\frac{T}{T_0}, 10^{-8} \right).$$

Apply a power-law schedule with exponent $\beta = 0.35$:

$$\text{range_limiter}_{\text{new}} = s^\beta.$$

This schedule starts with a large effective search radius when $T \approx T_0$ (global exploration) and smoothly shrinks the radius as the temperature cools (local refinement), replacing the simpler square-root schedule initially tested.

5. Experimental Results

5.1 Config

```
struct Config {
    // ----- time -----
    double time_limit_seconds = 220.0;

    // ----- temperature -----
    double min_temperature = 1e-5;

    // ----- optimal region probability -----
    double initial_region_prob = 0.5;
    double min_region_prob = 0.5;
    double max_region_prob = 0.9;
    double region_prob_step_up = 0.02;
    double region_prob_step_down = 0.02;

    // ----- moves per temperature -----
    int initial_moves_per_temp = 60000;
    int min_moves_per_temp = 40000;
    int max_moves_per_temp = 80000;
    double moves_scale_up = 1.2;
    double moves_scale_down = 0.8;

    // ----- range limiter (Previously window_scale) -----
    double initial_range_limiter = 1.0;
    double min_range_limiter = 0.01;
    double max_range_limiter = 1.0;

    // ----- Strategies -----
    // true: Use CalcOptimalRegion (median/quantile method)
    // false: Use CalcCenter (mean method + search radius)
    bool use_optimal_region_calc = false;

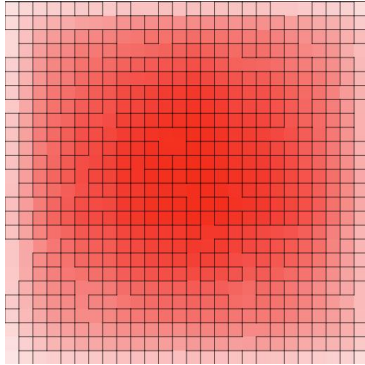
    // ----- random seed -----
    // 0 represents use random_device
    unsigned int random_seed = 7777;
};
```

5.2 Experimental result

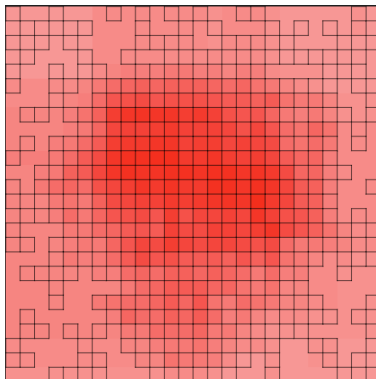
a.alu4

HPWL	Congestion coefficient
2497.00	1.066047590109

Initial



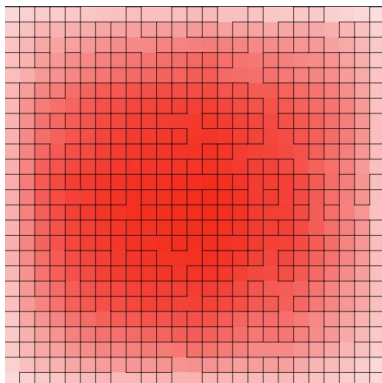
Final



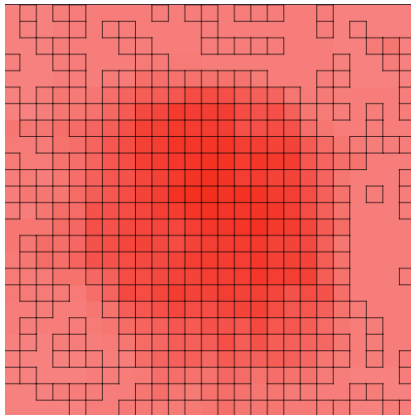
b.apex4

HPWL	Congestion coefficient
3775.00	1.039510574156

Initial



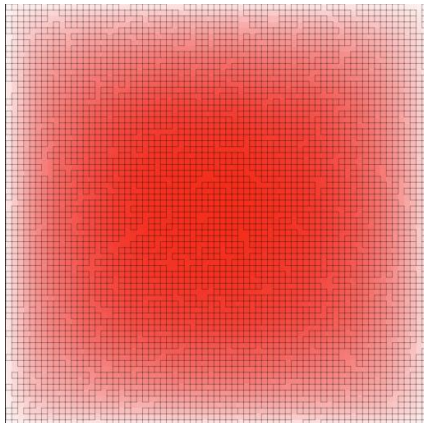
Final



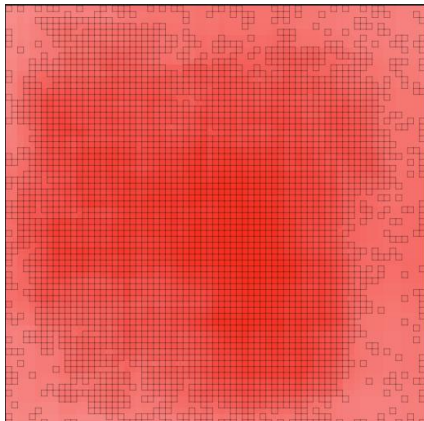
c.clma_2

HPWL	Congestion coefficient
24126.00	1.030619916740

Initial



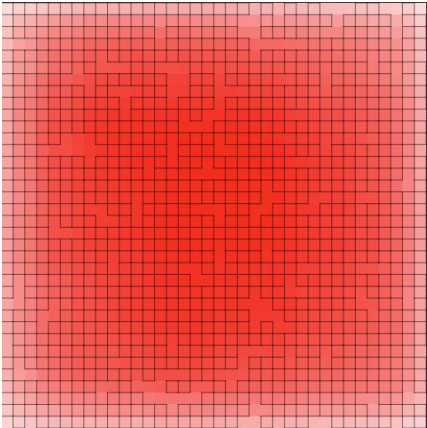
Final



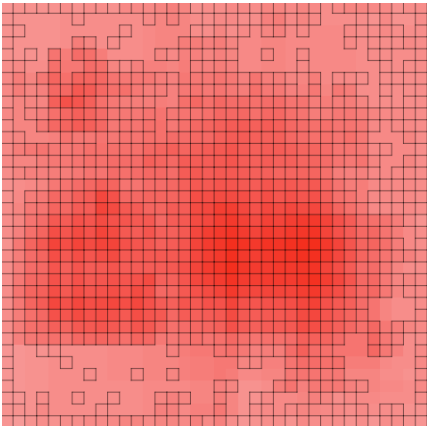
d.elliptic

HPWL	Congestion coefficient
4905.00	1.042681089363

Initial



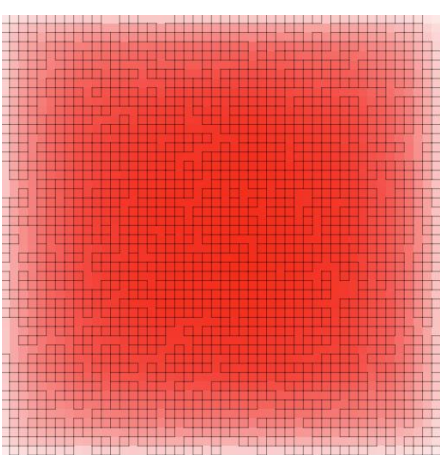
Final



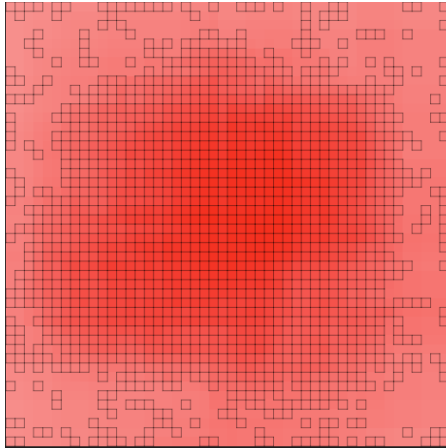
e.pdc

HPWL	Congestion coefficient
15004.00	1.046016003252

Initial



final



6. Conclusion

In this project, a simulated-annealing (SA) based congestion-driven placer for island-style FPGAs is implemented. Building on a VPR-like SA framework [2], the placer combines a traditional HPWL objective with a nets-overlap based congestion coefficient inspired by [1] and integrates them into a single scalar cost function. The algorithm starts from a simple random legal initial placement and then applies an SA loop with adaptive temperature, move budget, region probability, and move-range schedules, as described in the preceding sections.

On the algorithmic side, two main types of region-based moves are investigated. The first is an Optimal Region move that follows the median-based optimal region idea in [3]. The second is an averaging-based Average-Center Region (ACR) move, where the centers of the incident nets' bounding boxes are averaged to define an "ideal" target region. In the reported experiments, the ACR move provides more stable behavior and slightly better overall solutions than the optimal-region variant, and is therefore adopted as the default move operator. A k -power congestion model derived from [1], including an adaptive scheme in which the exponent depends on the size of congested areas, is also evaluated; however, the simpler and more robust $k = 1$ formulation is ultimately used for all reported results.

From an implementation perspective, the placer incorporates several

adaptive control mechanisms: automatic estimation of the initial temperature from sampled cost deltas, dynamic adjustment of the number of moves per temperature based on the observed acceptance rate, an adaptive region-move probability, and a power-law range limiter that smoothly shifts the search from global exploration to local refinement. These design choices make the SA process more self-tuning and reduce the need for manual parameter calibration across different benchmark circuits.

Overall, the implemented placer shows that a relatively simple SA framework, when combined with a nets-overlap congestion model and carefully designed region-based moves, can serve as a practical and extensible approach for congestion-aware FPGA placement.

7. References

- [1] "A Congestion Driven Placement Algorithm for FPGA Synthesis", In Proceedings of the International Conference on Field Programmable Logic and Applications (FPL 2006).
- [2] V. Betz and J. Rose, "VPR: A New Packing, Placement and Routing Tool for FPGA Research," in Proceedings of the 7th International Workshop on Field-Programmable Logic and Applications, 1997, pp. 213–222.
- [3] Rustam Zh. Chochaev, Vladislav A. Zhuravlev, Daniil A. Zheleznikov, "Simulated Annealing Based Placement in FPGAs with a Modified Permutation Operator and Congestion Optimization", *2024 IEEE 3rd International Conference on Problems of Informatics, Electronics and Radio Engineering (PIERE)*, pp.1190-1194, 2024.