Optimising Chip Placement Using Quantum Annealing

PL3001 Quantum Computing

Rishi Laddha, Siddharth Sahu, Suhani Agarwal, Tushar Goyal

1 The Problem

Chip placement is a critical and challenging aspect of the chip design process, involving the placement of a netlist graph of macros (like SRAMs) and standard cells (logic gates) onto a chip canvas. The goal is to optimize power, performance, and area (PPA) while adhering to constraints on placement density and routing congestion. This stage is complex due to the large size of netlist graphs, which can contain millions to billions of nodes, and the granularity of the placement grid.

Despite decades of research, the problem remains largely unsolved and requires human experts to iterate with existing placement tools over several weeks to meet design criteria. The complexity is further compounded by the high computational cost of evaluating target metrics, often taking hours or even days using industry-standard electronic design automation (EDA) tools. This makes the chip placement problem a significant bottleneck in the chip design cycle, necessitating innovative approaches to improve efficiency and effectiveness.

The paper we draw inspiration from, titled "Chip Placement with Deep Reinforcement Learning" by Mirhoseini et al. (2020), presents a sophisticated approach to chip placement using deep reinforcement learning techniques. The authors propose a reward function that captures the constraints inherent in chip placement, such as ensuring minimal wire length, reducing signal delay, and maintaining thermal balance. [Mir+20]

2 Our Approach

We want to optimise the placement of nodes (representing components or blocks) on a chip canvas taking into account power, performance and area (PPA) metrics. The objective is to minimize a weighted sum of wirelength and congestion while ensuring that the chip density stays within acceptable limits. This optimisation will be done using a process called Quantum Annealing on DWave's Quantum annealers.

To adapt our original problem to run on D-Wave's Quantum Computers, we can incorporate similar constraints and reward functions tailored to our specific problem domain. By formulating the problem as a QUBO and integrating constraints inspired by chip placement, we can leverage the capabilities of quantum annealers to explore a vast solution space and identify optimal configurations efficiently.

In brief, our approach to optimising chip placement using quantum annealing consists of the following steps -

1. **Formulating the problem as an objective function -** We represent the chip placement problem, along with the constraints we wish to impose, as a mathematical expression (called the *Objective (cost) function*). Our goal is to to find the lowest values that satisfy this function. These values represent good solutions to the chip placement problem.
2. **Find good solutions by sampling -** Sampling is the process by which values are sampled from low-energy states of the objective function which we defined earlier. We hope that an optimal solution will be sampled through this process.

We look at these steps in detail in the following section.

3 Quantum Annealing and Annealers

Quantum Annealing

Quantum annealing is a specialized process used by D-Wave quantum computers to find low-energy solutions for optimization and sampling problems. The process begins by initializing qubits in a superposition state, where they simultaneously represent multiple possible solutions. These qubits are then coupled to form an energy landscape corresponding to the problem's constraints and objective function.

As the annealing process progresses, the system gradually reduces quantum fluctuations, allowing the qubits to transition from higher energy states to lower energy states. This reduction is controlled by a time-dependent Hamiltonian, which guides the qubits toward the ground state of the problem Hamiltonian, representing the optimal solution.

Throughout this process, quantum tunneling enables qubits to overcome energy barriers that classical systems would find challenging, making quantum annealing particularly effective for complex optimization problems. The result is a highly efficient search for the global minimum in a vast solution space.

Quantum annealing leverages quantum phenomena such as entanglement and superposition to explore multiple solutions simultaneously, providing a significant advantage over classical approaches in specific applications like machine learning, material science, and logistics. [Sys23]

Problem Solving using Quantum Annealers

Problems submitted to the quantum computers have to be in binary quadratic model (BQM) format unconstrained with binary-valued variables and structured for the topology of the quantum processing unit (QPU). The binary quadratic model (BQM) class encodes Ising and quadratic unconstrained binary optimization (QUBO) models used by samplers such as the D-Wave system.

The QUBO model is an objective function of N binary variables represented as an upper-diagonal matrix Q, where diagonal terms are the linear coefficients and the nonzero off-diagonal terms the quadratic coefficients.

Once we are done formulating the objective function in the BQM format we pass them to samplers which explore the different possibilities encoded in the model, returning a list of variable assignments (solutions) prioritized towards those with lower energy (better solutions). We are using Hybrid samplers.

Samplers sample from low energy states of a problem's objective function, which is a mathematical expression of the energy of a system. A binary quadratic model (BQM) sampler, samples from low energy states in models such as those defined by an Ising equation or a QUBO problem and returns an iterable of samples, in order of increasing energy. (Transformations Between Ising and QUBO is trivial $s = 2x - 1$)

LeapHybridSampler is a class for using Leap's cloud-based hybrid BQM solvers. Leap's quantum-classical hybrid BQM solvers are intended to solve arbitrary application problems formulated as binary quadratic models (BQM).

Please refer to Figure 1 to understand how our code flow works with DWave Systems.

4 QUBO Formulation

Problem Formulation

For the D-Wave solver to find a solution by minimisation, we must express our problem as an objective function - a mathematical expression of the energy of the system. For most problems, the lower the energy of the objective function, the better the solution.

Nodes can not be added to the chip canvas haphazardly. They must follow a set of constraints to ensure optimal performance. Before we discuss what these constraints are, let us first look at how we incorporate

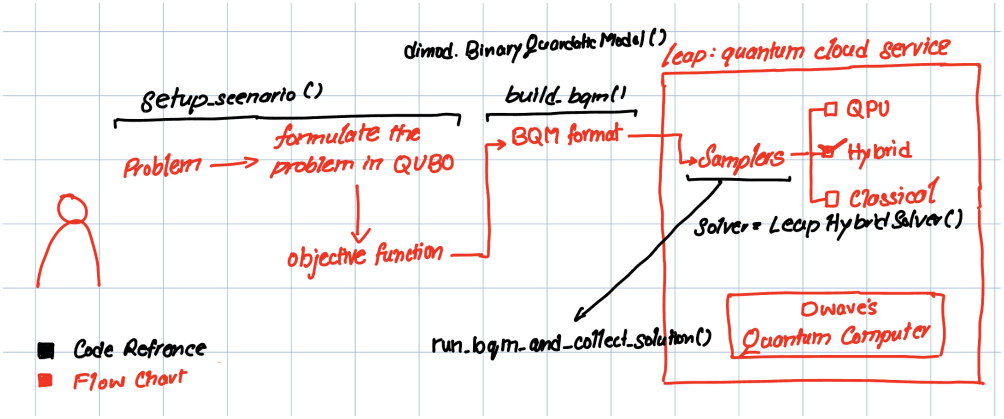

Figure 1: How DWave Works with the function calls in our code

these constraints in our objective function. Incorporating constraints involves penalizing solutions that violate those constraints. Hence, the objective function must be written such that it is minimized only when the constraints are satisfied, and gives large values when constraints are violated.

For the scope of this project, we work with the following three constraints [Mir+20] -

1. **Wirelength -** Wirelength between any two nodes on the chip canvas must be *minimised*. Each node on the canvas has a bias term representing its Manhattan distance to its connected nodes. These biases must be minimised.
2. **Congestion -** Congestion refers to the number of routes passing through a given region (say a predefined number of grid cells). Our goal is to *minimise* congestion as well.
3. **Density -** Density is the number of nodes present in a given region (again, a predefined number of grid cells). If there are too many nodes in a small region, this could lead to overheating of the component. Our task is to *minimise* density to avoid this issue.

Keeping these constraints in consideration, let us now construct the objective function to represent our problem in the QUBO form.

QUBO Representation

Our objective function (otherwise known as the QUBO representation of our problem) must take into consideration the constraints listed earlier. Let us construct objective functions for each constraint and put them together at the end for our final QUBO representation.

Wirelength Minimisation

$$L(Q_{ij}, x_i, x_j) = \sum_{(i,j)} Q_{ij} x_i x_j \tag{1}$$

Here, x_i, x_j are two nodes and Q_{ij} is the Manhatten distance between them. The function $L(Q_{ij}, x_i, x_j)$ is minimised when Q_{ij} is minimised, ensuring that the wirelength between nodes x_i and x_j is as less as possible.

Congestion Minimisation

$$C(x_r, x_s) = k \sum_{r \neq s} x_r x_s \tag{2}$$

Here, x_r, x_s represent two seperate routes passing through a given region. These take values 1 (if the route is present) or 0 (if not). k is a large arbitrary constant which gives a large value to the objective function $C(x_r, x_s)$ when both routes are present (i.e. $x_i = x_j = 1$). If only one route is present (i.e. either

x_i or x_j is 0), then the a product of 0 is added to the summation, thereby effectively reducing a multiple of k.

Density Minimisation

$$D(x_i) = \lambda \cdot \left[\max\left(0, \sum_i x_i - d\right)\right]^2 \tag{3}$$

Here, x_i represents each node present in a given region on the canvas and d is the maximum number of nodes permissible for that region. x_i can take values between 1 (if a node is present) and 0 (otherwise). λ is a large arbitrary If the number of nodes present in the region exceeds d, then the value of $D(x_i)$ becomes large. If the number of nodes is equal to or less than d, then $D(x_i)$ becomes 0 (because of the *max* function).

Final QUBO representation

Our final QUBO representation is the sum of equations (1), (2) and (3), as follows -

$$\begin{aligned} E(Q_{ij}, x_i, x_j; x_r, x_s; x_i) = L(Q_{ij}, x_i, x_j) \\ +C(x_r, x_s) \\ +D(x_i) \end{aligned} \tag{4}$$

Final QUBO representation
Hence, our final QUBO representation is as follows:

$$E(Q_{ij}, x_i, x_j; x_r, x_s; x_i) = \sum_{i,j} Q_{ij} x_i x_j + \sum_{\text{regions}} \left\{ k \cdot \sum_{r \neq s} x_r x_s + \lambda \cdot \left[\max\left(0, \sum_i x_i - d\right)\right]^2 \right\} \tag{5}$$

Our goal is to find solutions that minimise this QUBO representation of our problem.

Note A *region* in the case of QUBO formulation refers to a predefined cluster of grid cells (for example, a 4x4 cluster of grid cells).

5 Implementation

Link to Updated Jupyter Notebook

We read several papers and referred to the constraints that are needed to be taken care of while Chip Floorplanning and tried to formulate them as per the QUBO format required mathematically. We were able to come up with a sufficiently good representation emcompassing all the constraints. In the Project Update #2 we coded the QUBO with the following assumptions.

Assumption #1
All the macros (nodes) are connected to each other, whereas in a real chip floor planning problem not all components are connected to everything. We will try to encode this parameter later in our function for the final submission.

Assumption #2
In our QUBO formation we are trying to calculate how many routes are present in a particular region of interest. Since the routing will be done to minimize the Manhattan Distance, we are not considering if any existing macro (node) exists in the path that is used to calculate the Manhattan Distance. Neither are we counting the number of paths in a region of interest as suggested by our QUBO formulation.

In the final code implementation, we were successfully able to code the Assumption 1 as a constraint in the QUBO formulation, we tried to code the second assumption as well but we couldn't get to a feasible solution.

6 Code Implementation

Here we would like to discuss the main functions in our implementation of the QUBO.

Scenario Setup

`set_up_scenario` Function

```
def set_up_scenario(w, h, num_poi,
    ↪ num_existing_macros, netlist):
    G = nx.grid_2d_graph(w, h)
    nodes = list(G.nodes)

    # Add netlist connections
    G.add_edges_from(netlist)

    # Identify a fixed set of points of
        ↪  interest
    pois = random.sample(nodes, k=
        ↪ num_poi)

    # Identify a fixed set of current
        ↪ macro locations
    location_existing_macros = random.
        ↪ sample(nodes, k=
        ↪ num_existing_macros)

    # Identify potential new macro
        ↪ locations
    loc_pot_new_macros = list(G.nodes()
        ↪  - location_existing_macros)

    return G, pois,
        ↪ location_existing_macros,
        ↪ loc_pot_new_macros
```

Purpose: Initializes the grid graph and defines points of interest (POIs), existing macro locations, and potential new macro locations.

Parameters:

- `w` and `h`: Width and height of the grid.
- `num_poi`: Number of points of interest.
- `num_existing_macros`: Number of existing macro locations.
- `netlist`: List of edges representing connections between nodes.

Returns:

- `G`: The grid graph.
- `pois`: List of POI coordinates.
- `location_existing_macros`: List of existing macro coordinates.
- `loc_pot_new_macros`: List of potential new macro coordinates.

Distance Calculation

`manhattan_distance` Function

```
def manhattan_distance(a, b):
    return abs(a[0] - b[0]) + abs(a[1]
        ↪ - b[1])
```

Purpose: Computes the Manhattan distance between two points `a` and `b`.

Building the BQM

`build_bqm` Function

```
def build_bqm(loc_pot_new_macros,
    ↪ num_poi, pois,
    ↪ num_existing_macros,
    ↪ location_existing_macros,
    ↪ num_new_macros, netlist):
    gamma1 = len(loc_pot_new_macros) *
        ↪ 4
    gamma2 = len(loc_pot_new_macros) /
        ↪ 3
    gamma3 = len(loc_pot_new_macros) *
        ↪ 1.7
    gamma4 = len(loc_pot_new_macros) **
        ↪  3

    bqm = dimod.BinaryQuadraticModel(
        ↪ len(loc_pot_new_macros), '
        ↪ BINARY')

    if num_poi > 0:
        for i in range(len(
            ↪ loc_pot_new_macros)):
            cand_loc =
                ↪ loc_pot_new_macros[i
                ↪ ]
            avg_dist = sum(
                ↪ manhattan_distance(
                ↪ cand_loc, loc) for
                ↪ loc in pois) /
                ↪ num_poi
```

```
            bqm.linear[i] += avg_dist *
                ↪  gamma1

    if num_existing_macros > 0:
        for i in range(len(
            ↪ loc_pot_new_macros)):
            cand_loc =
                ↪ loc_pot_new_macros[i
                ↪ ]
            avg_dist = -sum(
                ↪ manhattan_distance(
                ↪ cand_loc, loc) for
                ↪ loc in
                ↪ location_existing_macros
                ↪ ) /
                ↪ num_existing_macros
            bqm.linear[i] += avg_dist *
                ↪  gamma2

    if num_new_macros > 1:
        for i in range(len(
            ↪ loc_pot_new_macros)):
            for j in range(i+1, len(
                ↪ loc_pot_new_macros))
                ↪ :
                ai = loc_pot_new_macros
                    ↪ [i]
                aj = loc_pot_new_macros
                    ↪ [j]
                dist = -
                    ↪ manhattan_distance
                    ↪ (ai, aj)
                bqm.add_interaction(i,
                    ↪ j, dist * gamma3
                    ↪ )

    for (i, j) in netlist:
        if i in loc_pot_new_macros and
            ↪ j in loc_pot_new_macros:
            idx_i = loc_pot_new_macros.
                ↪ index(i)
            idx_j = loc_pot_new_macros.
                ↪ index(j)
            bqm.add_interaction(idx_i,
                ↪ idx_j, -
                ↪ manhattan_distance(i
                ↪ , j) * gamma3)

    bqm.update(dimod.generators.
        ↪ combinations(bqm.variables,
        ↪ num_new_macros, strength=
        ↪ gamma4))

    return bqm
```

Purpose: Constructs a Binary Quadratic Model for the optimization problem from QUBO using APIs offered by DWave.

Parameters:

- `loc_pot_new_macros`: Potential new macro locations.
- `num_poi`: Number of POIs.
- `pois`: List of POI coordinates.
- `num_existing_macros`: Number of existing macro locations.
- `location_existing_macros`: List of existing macro coordinates.
- `num_new_macros`: Number of new macros to be placed.
- `netlist`: List of edges representing connections between nodes.

Returns:

- `bqm`: The constructed BQM.

Running the BQM and Collecting Solutions

`run_bqm_and_collect_solutions` **Function**

```
def run_bqm_and_collect_solutions(bqm,
    ↪ sampler, loc_pot_new_macros, **
    ↪ kwargs):
    sampleset = sampler.sample(bqm,
        ↪ label='Example - Macro
        ↪ Placement', **kwargs)
    ss = sampleset.first.sample
    new_macro_nodes = [
        ↪ loc_pot_new_macros[k] for k,
        ↪  v in ss.items() if v == 1]
    return new_macro_nodes
```

Purpose: Solves the BQM using the provided sampler and returns the new macro locations.

Parameters:

- `bqm`: The BQM for the problem.
- `sampler`: The sampler or solver to be used.
- `loc_pot_new_macros`: Potential new macro locations.

Returns:

- `new_macro_nodes`: List of new macro coordinates.

7 Results

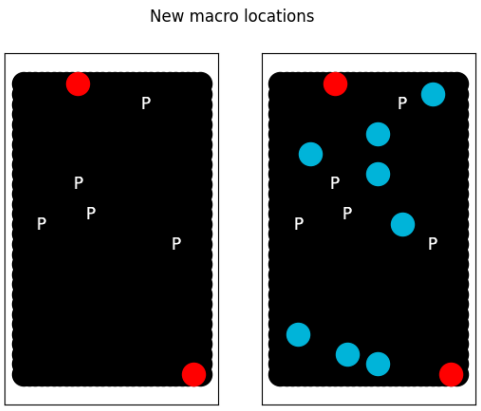

Figure 2: Red - Existing Macros, P - Potential Locations, Blue - Actual Placement of New Macros

```
Solution returned:
New macro locations: [(25, 28), (3, 4),
    ↪  (5, 22), (20, 15), (11, 2),
    ↪ (16, 20), (16, 24), (16, 1)]
Average distance to POIs: [21.4,
    ↪ 24.599999999999998, 15.4, 12.2,
    ↪ 22.6, 12.4, 14.8, 24.6]
Average distance to old macro: [25.0,
    ↪ 21.0, 7.0, 15.0, 19.0, 8.0,
    ↪ 12.0, 25.0]
Distance between new macros: 602
```

During the course of this project, we were able to identify a current Quantum Computing Technology available (DWave's Quantum Annealers) and map it to a constraint satisfaction problem for Floor Planning in the Chip Domain. The dominant way that it happens is either manually or by Deep Reinforcement as shown by Google. [Mir+20] We took inspiration for the constraints from the same deep reinforcement paper, and designed a QUBO formulation capturing the required constraints but leaving out one due to it's complex nature of implementation. The sampler was able to return viable results and we can can keep sampling it again and again till we reach to our desired solution. The journey was really fun, and maybe the results might not be ground-breaking but we hope as the Quantum Computing Technology advances, we can exploit it more for the algorithms that can benefit from running in parallel. Thanks to Prof Nitin and Viraj for offering the course and continued involvement and valuable feedback, that helped us complete the project.

References

[Mir+20] Azalia Mirhoseini et al. "Chip Placement with Deep Reinforcement Learning". In: *CoRR* abs/2004.10746 (2020). arXiv: 2004.10746. URL: https://arxiv.org/abs/2004.10746.

[BOH23] BOHR TECHNOLOGY. *Quantum Solutions for the Traveling Salesman Problem*. https://github.com/BOHRTECHNOLOGY/quantum_tsp. [Online; accessed 21-March-2024]. 2023.

[Fap23] Faptimus420. *Quantum Route Optimizer*. https://github.com/Faptimus420/QuantumRouteOptimizer. [Online; accessed 15-March-2024]. 2023.

[Inc23] D-Wave Systems Inc. *D-Wave Examples: EV Charger Placement*. https://github.com/dwave-examples/ev-charger-placement. 2023.

[mst23] mstechly. *Quantum TSP Tutorials*. https://github.com/mstechly/quantum_tsp_tutorials. [Online; accessed 21-March-2024]. 2023.

[Sys23] D-Wave Systems. *What is Quantum Annealing?* https://docs.dwavesys.com/docs/latest/c_gs_2.html. Accessed: 2024-05-25. 2023.

[Wik24] Wikipedia contributors. *Quadratic unconstrained binary optimization*. [Online; accessed 15-March-2024]. 2024. URL: https://en.wikipedia.org/wiki/Quadratic_unconstrained_binary_optimization.