International Journal of Networking and Computing – www.ijnc.org, ISSN 2185-2847
Volume 16, Number 1, pages 2–19, January 2026

RL-driven Annealing Computation for QUBO on Multi-GPU System

Reo Gakumi
Graduate School of Informatics, Kyoto University
Yoshida Honmachi, Sakyo-ku, Kyoto, 606-8501, JAPAN

Ryota Yasudo
Academic Center for Computing and Media Studies, Kyoto University
Yoshida Honmachi, Sakyo-ku, Kyoto, 606-8501, JAPAN

Received: July 30, 2025
Accepted: September 2, 2025
Communicated by Susumu Matsumae

Abstract

Quadratic Unconstrained Binary Optimization (QUBO) has emerged as a unifying framework for diverse NP-hard combinatorial optimization problems. However, a major challenge in existing QUBO solvers is the need for extensive manual tuning of algorithmic hyperparameters, such as temperature schedules in simulated annealing, which can vary greatly in effectiveness depending on the problem instance. In this paper, we propose RL-driven annealing (RLA), a novel approach that integrates reinforcement learning (RL) with annealing-based local searches. Rather than relying on hand-crafted heuristics, RLA trains an agent to adaptively determine a bit-flip policy by observing the statistical properties of the energy differences in the objective function. Crucially, RLA encodes QUBO states as fixed-dimensional statistics, making the method scalable to various problem sizes. To further support large-scale problems, we employ distributed training on multi-GPU platforms using JAX's pmap, parallelizing both environment simulation and policy updates. Experimental evaluations on benchmark datasets, including TSP, QAP, Max-Cut, and randomly generated QUBO instances, demonstrate that RLA achieves solution qualities on par with or better than conventional annealing-based methods, while maintaining robust performance across diverse problem instances without extensive hyperparameter tuning. These results highlight RLA as a promising step toward a flexible and practical solver for QUBO-based applications.

1 Introduction

Annealing computation is a heuristic local search process commonly employed to solve NP-hard combinatorial optimization problems, whose name is derived from a common metaheuristics called simulated annealing. Recently, it is used in conjunction with quantum or quantum-inspired Ising machines, or equivalently quadratic unconstrained binary optimization (QUBO) solvers. Given the NP-hard nature of QUBO problem, numerous quantum-inspired solvers have been developed by exploiting parallel computation of classical computers, including GPUs [1,2], FPGAs [3,4], ASICs [5–7], and even emerging technology [8]. Interestingly, such classical methods are reported to outperform current quantum annealing methods because of the impact of noise, in terms of solution quality and the size of the problems to solve [9]. Consequently, the development of annealing computation has

a potential to be a unified solver for a wide range of combinatorial optimization problems in the society. Once a problem is reduced to a QUBO form, we can obtain semi-optimal solutions within reasonable timeframes without the need to design specialized solvers for each specific problem.

However, although QUBO solvers have such potential, using them as a unified solver is practically difficult. This is because annealing computation has a large number of parameters such as temperature scheduling. Obviously, the best parameter setting depends on problem instances, and hence we must practically try to adjust the parameters for obtaining good solutions. This results in several issues. First, adjusting parameters takes a lot of time, which should be much larger than the execution time of annealing computation itself. Second, it is impossible in principle to determine the best parameter setting. We typically make only simple setting. For example, the temperature settings can be configured to decrease linearly. If there is a more complex but better setting, we should find it, as it could lead to better solutions. It would also be interesting to see a complex but better setting and how it changes according to problem instances.

In this study, we propose a new concept of RL-driven annealing (RLA), where the parameters of a local search are adjusted by reinforcement learning (RL). RL has recently been used to solve combinatorial optimization problems, as well as playing board games and finding new algorithms. Solving QUBO with RL has also been proposed, but the solution quality is limited. Instead, our concept uses RL to adjust parameters of heuristic algorithms, leading to much better solutions. Our proposed method is a hybrid method combining conventional heuristic algorithms and RL, and the focus of this study is on QUBO.

RLA should firstly train a model, and then it can infer a good parameter setting. Regarding this concept, we can consider multiple strategies in terms of how training and inference are used. In an ideal strategy, we may train a model from a wide range of problem instances and then carry out an inference for any problem instance. If this strategy is feasible, it would mean that we obtain a truly unified model. However, this strategy is difficult to design in practice. This is because it would require tremendous training data, and additionally it is unclear whether such a universal method for parameter setting exists. Next, the second possible strategy trains a model for some similar instances, e.g., those of a specific original problem before converting to QUBO. Then, a model infers parameters for similar problem instances. This second strategy is more realistic than a unified model, but it is still highly challenging. Thus, this paper focuses on the third strategy, where we train a model with only one specific problem instance and obtain a solution at the same time. In other words, a training process corresponds to a search process instead of an inference process. It should obtain better solutions than the first and second strategies obtain, while it takes larger time.

The primary contributions of this study can be summarized as follows. We propose a RL-based local search approach for QUBO, the RL-driven annealing (RLA). This approach is the first method that applies RL to adjust parameters of annealing computation. Among some possible strategies above, this study focuses on a strategy where a neural network model searches for solutions while a training process. Furthermore, we implement our proposed method on GPU and provide parallel training.

In Section 2, we introduce QUBO, typical annealing computation, and reinforcement learning. Section 3 introduces the RL-driven annealing starting from its concept and describes the RL-environment setting. In Section 4, the model architecture is explained. We detail the implementation on a multi-GPU system in Section 5. Section 6 evaluates the performance of the RL-driven annealing using several types of QUBO instances. Finally, Section 7 presents conclusions.

2 Background and Motivation

2.1 QUBO and annealing computation

The quadratic unconstrained binary optimization problem (QUBO) is an NP-hard combinatorial optimization problem. An instance of the QUBO problem is represented by a $N \times N$ symmetric matrix $Q = (q_{i,j})$ $(0 \leq i, j \leq N-1)$, and a decision variable vector is an N-bit vector $\boldsymbol{x} =$

$(x_0, x_1, \cdots, x_{N-1})$ $(x_i \in \{0, 1\})$. The objective function is the quadratic function $f(\boldsymbol{x})$ defined by

$$f(\boldsymbol{x}) = \boldsymbol{x}^{\mathrm{T}} Q \boldsymbol{x} = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} q_{i,j} x_i x_j.$$

Clearly $f(\boldsymbol{x})$ includes linear terms $q_{0,0}x_0, q_{1,1}x_1, q_{2,2}x_2, \cdots$ (note that ${x_i}^2 = x_i$ holds because x_i is 0 or 1) and quadratic terms $q_{i,j}x_i x_j$ $(i \neq j)$. The QUBO formulations of many combinatorial optimization problems have been proposed thus far [10].

For QUBO problems, the annealing computation tries to find the optimal solution, i.e., a solution with minimum $f(\boldsymbol{x})$, by using a local search algorithm. Local search algorithm starts from an initial solution, and iteratively moves from a current solution to a neighboring solution in the search space so that the objective function improves. For the QUBO, neighboring solutions are typically defined by a solution where one decision variable is flipped. For example, when $N = 4$, neighbor solutions of $(0, 0, 0, 0)$ is $(0, 0, 0, 1)$, $(0, 0, 1, 0)$, $(0, 1, 0, 0)$, and $(1, 0, 0, 0)$. In this situation, it is computationally effective to make an initial solution a zero vector $(0, 0, 0, 0)$, because its objective function is clearly 0, and the objective functions of the neighboring solutions are $q_{0,0}, q_{1,1}, \cdots, q_{N-1,N-1}$, respectively.

Typical annealing computation stores the differences of the objective functions when each decision variable is flipped, which are denoted by $\Delta_0, \Delta_1, \cdots, \Delta_{N-1}$. Thus, for an initial solution $(0, 0, 0, 0)$, $\Delta_i = q_{i,i}$ holds for $i = 0, 1, 2, 3$. An incremental computation enables the update of Δ_i in $O(1)$ time [2], and thus local search algorithm works fast. From the above, the search process of typical annealing computation can be written as follows.

1. It starts from an initial solution $\boldsymbol{x} = (0, 0, \cdots, 0)$ and set $\Delta_i = q_{i,i}$ for $i = 0, 1, \cdots, N-1$.

2. It selects a decision variable to be flipped according to the values of Δ_i.

3. It flips the selected decision variable, updates the values of Δ_i, and returns to Step 2).

The search process is illustrated in Fig. 1 (a). We can say that the essence of the annealing computation is the policy of Step 2), which we call the Δ *based flip policy*.

One simple Δ-based flip policy may select a decision variable with the minimum value of Δ. This policy, corresponding to a hill climbing algorithm, can find a local minimum solution quickly; however, it may not find a global optimal solution. To find a global solution, a number of more sophisticated policies have been proposed.

In this study, we focus on the *threshold accepting* (*TA*) policy. The TA sets a threshold and compares it with each value of Δ; if Δ_i is less than or equal to a threshold, the corresponding decision variable x_i becomes a flip candidate. Then it selects a bit randomly from the flip candidates. Two enhancements to this method are the Max-Min TA and PositiveMin-Min TA, which are proposed in [4]. The bits of the decision variable vector $\boldsymbol{x}$ are selected randomly, and a local search is repeatedly performed where bits are flipped with a probability determined by the temperature parameter. However, the bit flip is not necessarily executed in every iteration. In contrast, the two algorithms from related research guarantee bit flipping regardless of the temperature scheduling, while attempting to escape local optima and achieving high-speed implementation on FPGA. The literature indicates that the Max-Min TA algorithm is suitable for the Max-Cut problem, while the PositiveMin-Min TA algorithm is effective for the Traveling Salesman Problem (TSP). In the Max-Min TA, a threshold is set between the maximum Δ, denoted by $\Delta_{\max}$, and the minimum Δ, denoted by $\Delta_{\min}$. Specifically, a threshold is $\Delta_{\min} + T(\Delta_{\max} - \Delta_{\min})$, where T is a parameter in the range of $[0, 1]$. On the other hand, in the PositiveMin-Min TA, a threshold is set to be the minimum value of Δ where $\Delta > 0$. If there is no positive Δ, a threshold becomes 0.

Although both methods work well, the quality of the solution depends on the problem instance. In [4], it is reported that the Max-Min TA is not well-suited for the traveling salesman problem (TSP), and the PositiveMin-Min TA is not well-suited for the Max-Cut problem. Furthermore, the best setting of T is unclear for the Max-Min TA. We must carefully adjust the parameter, which should take a long time. Our study builds upon the TA and addresses the parameter setting issue.

2.2 Reinforcement learning

Reinforcement learning (RL) [11] is one of the major machine learning paradigms. It enables an agent to learn a policy that maximizes cumulative rewards by repeatedly interacting with the environment. Specifically, the agent observes the state of the environment, takes an action, receives a reward, and transitions to a new state. Although a simple RL algorithm, such as Q-learning, estimates the action-value function (Q-values) for each state–action pair, it becomes impractical when the state–action space is large. To address this issue, deep neural networks can be used to approximate these values, leading to the framework known as deep reinforcement learning (deep RL). Our proposed method builds upon deep RL.

With the advancement of machine learning techniques, deep RL has already outperformed humans in domains such as board games and robotics. More recently, it has been applied to broader tasks, including the exploratory discovery of matrix multiplication algorithms and the optimization of sorting algorithms at the assembly-instruction level [12]. For solving combinatorial optimization problems, RL methods have been successfully applied to the traveling salesman problem [13], the maximum cut problem [14], and the vehicle routing problem [15]. Once a neural network model is trained, we can infer a semi-optimal solution faster than meta-heuristics. Previous studies have employed the multi-armed bandit problem, which is a simple reinforcement learning, for enhancing QUBO solvers [16]. However, this approach remains relatively simple, limiting its potential for improving solutions. Instead, we leverage deep RL for QUBO to enable more effective learning of parameter scheduling.

3 RL-driven annealing

3.1 Proposed concept and methodology

In this study, we introduce a new framework called *RL-driven Annealing* (RLA), which addresses the limitations of conventional annealing-based QUBO solvers by integrating deep reinforcement learning (DRL). Traditional approaches, such as threshold accepting, typically require manual tuning of hyperparameters such as temperature schedules or acceptance thresholds. These methods can handle a range of QUBO instances, but face two major challenges: (1) the optimal parameter settings vary substantially across different problem types and sizes, and (2) scaling up to larger QUBO instance increases computational overhead and makes parameter selection more complex.

RLA addresses these issues by replacing hand-crafted bit-flip policies with a DRL agent capable of adaptively learning which bits to flip. The agent uses an abstracted representation of the energy-difference distribution (denoted by Δ) and outputs a continuous action $a_t \in [0, 1]$. This action is mapped to a threshold that partitions the interval $[\Delta_{\min}, \Delta_{\max}]$; bits whose Δ_i values fall within or below this threshold are candidates for flipping, and one candidate is chosen at random. Through iterative trial-and-error, the agent learns how aggressively to flip bits, balancing exploration with exploitation and automatically adapting to diverse QUBO structures.

To further enhance scalability, we incorporate distributed reinforcement learning in multi-GPU systems using JAX's `pmap` functionality. Parallelizing the environment both increases computational throughput and diversifies the range of candidate flips observed by the agent. As a result, learning progresses more rapidly, and the quality of the solution for larger QUBOs improves.

3.2 State abstraction

Within the QUBO framework, a candidate solution can be represented by a decision variable vector $\boldsymbol{x}$ of length N (i.e., $\boldsymbol{x} \in \{0, 1\}^N$). As the size of the vector increases, the number of possible combinations grows exponentially, resulting in 2^N configurations for a vector of length N, and the size of an action space of each state is N. This exponential growth makes direct handling of Δ computationally impractical.

To make the problem tractable for reinforcement learning, we adopt an abstracted environment representation that remains the same dimensionally for any N. Rather than providing each bit's raw

energy difference Δ to the agent, we compute fixed-dimensional statistical summaries (e.g., mean, variance, quantiles) of the normalized Δ values. These statistics form the agent's observation at every step.

The DRL policy network then produces a continuous action that reflects how 'aggressively' to accept flips. In concrete terms, the action is assigned to a threshold within the range $[\Delta_{\min}, \Delta_{\max}]$, and a bit from among those below this threshold is chosen to flip. This abstraction ensures that, regardless of the QUBO size, the agent observes a consistent set of features and works within a fixed continuous action interval. Consequently, the same network architecture can be applied across a variety of problem scales and structures without redesign.

By capturing the essential distribution of energy differences, this approach facilitates both effective exploration (sometimes flipping worse bits to escape local optima) and exploitation (tending to flip promising bits). As we demonstrate in our experiments, the combination of DRL-based bit-flip policies and an abstracted state/action representation enables RLA to deliver robust performance across multiple problem types and sizes without extensive hyperparameters tuning.

3.3 RL-environment state setting

For the RLA, we define the state space $\mathcal{S}$, the action space $\mathcal{A}$, the state transition $\mathcal{T}$, and the reward function $\mathcal{R}$ as follows. Figure 2 and Algorithm 1 illustrates how these components interact within our reinforcement learning framework.

3.3.1 State $\mathcal{S}$

We establish a scalable environment setting independent of the size of QUBO instances, in which the decision variable vector $\boldsymbol{x}$ is not provided as an observable state directly. In this environment, the observable state reflecting the nature of Δ_t consists of five elements as follows.

- The proportion of positive Δ.
- The proportion of positive $\boldsymbol{x}$.
- The standard deviation of Δ.
- The mean of Δ.
- The interquartile range of Δ.

These basic statistical inputs contain various information useful for the solution search. For example, the proportion of positive Δ can be interpreted as the fraction of actions in the next state that are expected to worsen the solution in the short term.

3.3.2 Action $\mathcal{A}$

The action corresponds to a flip of a decision variable. It is determined by a continuous value $a_t \in [0, 1]$. Based on the value of a_t, the threshold for selecting elements to flip is calculated as follows:

$$\mathbf{th} = \Delta_{\min} + a_t(\Delta_{\max} - \Delta_{\min}),$$

where a decision variable x_i to be flipped are randomly chosen from i that satisfies $\Delta_i \leq \mathbf{th}$ for $0 \leq i < N$. When $a_t = 0$, the threshold corresponds to $\Delta_{\min}$. When $a_t = 1$, it corresponds to $\Delta_{\max}$, ensuring at least one element is below the threshold at each step.

Since Δ represents the short-term favorability of actions for transitioning to the next state, the value of action a_t can be interpreted as a continuous measure of how much deterioration in the short-term state transition is acceptable. The best move that decreases the objective function is scaled to 0, and the worst move that increases it is scaled to $+1$ in the normalized Δ'. Thus, the action a_t indicates the threshold for accepting unfavorable transitions. The conversion from the continuous action selected to a discrete action, which corresponds to flipping one element of the decision variable

vector, is defined by randomly selecting a candidate element that meets the condition and flipping its corresponding decision variable. This mapping from state-action design is shown by the random selection of an element among the suitable candidates for flipping.

3.3.3 State Transition $\mathcal{T}$

The agent performs a probabilistic state transition based on the continuous value provided as input. The agent then receives the statistical summary S_t of the next state as an observation. Since the goal of this framework is to find better solutions even after prolonged exploration, the episode does not terminate until the agent reaches a local optimal solution.

Specifically, the episode continues until the decision variable vector $\mathbf{x}_t$ reaches a local optimum, which is defined as the point where all Δ_i values are greater than 0. Once this condition is met, the environment is reset and all elements in the initial state are set back to 0. This mechanism ensures that the agent can continue exploring until it finds a solution that cannot be improved further, facilitating longer-term exploration and avoiding premature convergence to suboptimal solutions.

3.3.4 Reward $\mathcal{R}$

In this study, we design the reward function to maintain a consistent magnitude across different sizes of QUBO instances and varying absolute values of matrix elements. In the QUBO, the input matrix Q can include large values, especially when the constraint terms dominate, which can lead to extremely large values of the objective function. Such large values can destabilize the learning process when the RL assigns excessively high rewards. To counteract this, reward clipping [17] is applied, restricting the reward within the range of $[-1, 1]$. By capping the rewards, we prevent a rapid increase in Q-values, stabilizing the gradient updates and ensuring a smoother learning process.

Given that the scale of rewards can vary significantly between tasks, we apply a simple clipping strategy: any positive reward is capped at 1, and any negative reward is capped at -1, while a reward of 0 remains unchanged. This approach not only limits the magnitude of the error derivatives, but also allows for a consistent learning rate across different environments. However, it is important to note that clipping rewards may reduce the agent's ability to distinguish between rewards of different magnitudes, potentially affecting performance [18].

In this specific environment, we aim to build an agent that focuses on finding better solutions, even if it requires more time. Therefore, the discount factor is set to $\gamma = 1.0$. Typically, the cumulative reward $G(\tau)$ of trajectory τ is defined as $G(\tau) = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots = \sum_{k=1}^{\infty} \gamma^{k-1} r_{t+k}$, where r_t is the reward at time t. However, when $\gamma = 1.0$, the cumulative reward simply becomes the sum of rewards over time: $G_t = \sum_{k=1}^{\infty} r_{t+k}$.

To normalize the QUBO matrix Q, we scale the matrix by dividing each element by the largest absolute value in the matrix. This results in a symmetric matrix Q' where the maximum element is 1. If we denote by Δ' the change in the objective function computed using matrix Q', and by Δ the change computed using the original matrix Q, the objective function can be written as follows:

$$\begin{aligned} f(\boldsymbol{x}) &= \boldsymbol{x}^{\mathrm{T}} Q \boldsymbol{x} = \sum_{k=1}^{\infty} \Delta_{t+k} \\ &= \texttt{absmax}(Q) \times \boldsymbol{x}^{\mathrm{T}} Q' \boldsymbol{x} \\ &= \texttt{absmax}(Q) \times \texttt{matrix\_size}(Q) \times \sum_{k=1}^{\infty} \Delta'_{t+k}. \end{aligned}$$

This formulation ensures that rewards remain normalized, preventing any single element from dominating the learning process.

4 Model architecture

We employ the Proximal Policy Optimization (PPO) algorithm, which is implemented using the JAX and Flax frameworks. It trains an agent in an environment where episodes continue until

Algorithm 1 Proposed RL-environment state setting

Require: $\boldsymbol{W}$: weight matrix of the QUBO Problem
Ensure: $\boldsymbol{B}, e_B$: the best solution $\boldsymbol{B}, e_B$: the best energy

Initialization:
  $\boldsymbol{W'} \leftarrow \boldsymbol{W} / \text{absmax}(\boldsymbol{W})$ ▷ *Normalized QUBO Matrix*
  $\boldsymbol{X} \leftarrow 00\cdots0$ ▷ *Initial vector of* $\boldsymbol{x}$
  $\boldsymbol{B} \leftarrow 00\cdots0$ ▷ *Discovered optimal vector of* $\boldsymbol{x}_B$
  $e_B \leftarrow 0$ ▷ *Discovered optimal energy* $E(\boldsymbol{x}_B)$
  $\Delta \leftarrow \text{diag}(\boldsymbol{W'})$
loop
  $k \leftarrow$ AGENT_SELECT$(\Delta, \boldsymbol{x})$
  $\boldsymbol{X} \leftarrow$ FLIP$(\boldsymbol{X}, k)$
  $e \leftarrow e + \Delta_k$
  $\Delta \leftarrow$ CALCULATE_DELTA$(\boldsymbol{X}, \boldsymbol{W'}, \Delta, k)$
  if $e < e_B$ **then** ▷ *update the best solution*
    $B \leftarrow X$
    $e_B \leftarrow e$
function CALCULATE_STAT$(\Delta, \boldsymbol{x})$
  $delta\_p\_num =$ COUNT$(\Delta > 0)$
  $delta\_positive; stats[0] \leftarrow delta\_p\_num / \text{size}(\Delta)$
  $vec\_positive; stats[1] \leftarrow \Sigma\boldsymbol{x} / \text{size}(\Delta)$
  $delta\_std; stats[2] \leftarrow$ STANDARDDEVIATION(Δ)
  $delta\_mean; stats[3] \leftarrow$ MEAN(Δ)
  $delta\_iqr; stats[4] \leftarrow$ INTERQUARTILERANGE(Δ)
  $obs \leftarrow$ CREATEARRAY$([stats])$
  return *obs*
function AGENT_SELECT$(\Delta,\boldsymbol{x})$
  $s_t \leftarrow$ CALCULATE_STAT$(\Delta, \boldsymbol{x})$
  $a_t \leftarrow$ Inference from Agent model with s_t ▷ *Action* $a_t \in [0, 1]$
  $th \leftarrow min(\Delta) + a_t(max(\Delta) - min(\Delta))$
  randomly choose $k \in \{i | d_i \leq th (0 \leq i < n)\}$
  return k
function FLIP$(\boldsymbol{x},k)$
  $\boldsymbol{x}_k \leftarrow 1 - \boldsymbol{x}_k$
  return $\boldsymbol{x}$
function CALCULATE_DELTA$(\boldsymbol{x}, W, \Delta, k)$
  $\sigma(x) = 2x - 1$ ▷ $\sigma : \{0, 1\} \rightarrow \{-1, +1\}$
  for $i = 0 \rightarrow N - 1$ **do**
    if $k = i$ **then**
      $\Delta_i \leftarrow -\Delta_i$
    else
      $\Delta_i \leftarrow \Delta_i + 2W_{i,k}\sigma(x_i)\sigma(x_k)$
  return $\boldsymbol{\Delta}$

a local optimum is reached. Our primary objective is to enable this agent to gradually increase long-term profits, regardless of the time required. At each step of the task, the agent receives a reward equal to the incremental change in an objective function from the previous step, so that the cumulative sum of rewards corresponds directly to the objective function itself.

To ensure computational efficiency and scalability, we leverage data parallelism across multiple GPUs using JAX's `pmap` function. This approach synchronizes model updates among all devices, thereby improving training throughput as the number of GPUs increases. While multi-GPU configurations can introduce communication overhead, particularly in gradient synchronization, the resulting speedup and capacity for larger batch processing outweigh these costs in most practical settings.

In this framework, we adopt a standard Actor-Critic algorithm. The actor (a policy network) and the critic (a value network) share an initial feature extractor to reduce redundant computation and to capture domain-relevant features more effectively. The feature extractor processes each observation vector $\mathbf{x} \in \mathbb{R}^n$ through two fully connected layers, each of which projects its input to a 256-dimensional representation followed by a hyperbolic tangent activation.

All network weights are initialized using an orthogonal scheme to promote stable gradients; hidden layers adopt $\sqrt{2}$ as the gain, whereas the actor's final layer uses a lower gain (e.g., 0.01) to

avoid overly large initial actions and to encourage more effective early exploration.

We present detailed comparisons of single-GPU and multi-GPU training modes to elucidate how model updates differ between these setups. In the single-GPU mode, gradients are computed and applied on a single device, which simplifies implementation but may limit the rate of experience processing. By contrast, the multi-GPU training with `pmap` enables parallel gradient computations across devices and aggregates them before performing parameter updates. Such an approach accelerates learning, especially in environments with high-dimensional inputs or large batch sizes. However, fully leveraging these benefits may require careful tuning of batch splitting and communication strategies.

In summary, our system architecture integrates a well-established Actor-Critic network design with scalable multi-GPU training using JAX's `pmap` function, ensuring both robust learning dynamics and computational efficiency. By leveraging data parallelism and an environment with unlimited episode lengths, our approach enables the agent to consistently converge toward locally optimal behaviors while maximizing returns based on a continuously evolving objective function.

5 Implementation on a multi-GPU system

We use JAX library, which is an open-source Python framework designed for high-performance numerical computing. It transforms Python functions into operations on GPU or TPU hardware, supporting automatic differentiation and just-in-time compilation, which makes it particularly well-suited for machine learning tasks [19]. Our code builds on the PPO implementation from purejaxrl [20] and integrates gymnax-based environments [21] for consistent interface between QUBO optimization and standard reinforcement learning tasks. By utilizing JAX-based reinforcement learning, we can perform computations efficiently on GPUs, significantly accelerating the training process while maintaining flexibility in model development. In multi-GPU mode, the `pmap` function from JAX enables parallel computations and collective operations across devices. Figure 3 provides an overview of Multi-RLA workflow with three GPUs. Note that in the case of Single-RLA, no data transfer between GPUs is required; consequently, its workflow is essentially equivalent to a subset of the Multi-RLA approach.

Our approach integrates an Actor-Critic framework with Proximal Policy Optimization (PPO), wherein each device (or GPU) interacts with multiple parallel environments to collect transitions, compute policy/value gradients, and synchronize model parameters. In the figure, blue arrows denote the flow of data (e.g., observation vectors, rewards, and environment states), while red arrows denote gradient flow and parameter updates. At the left block, QUBO matrices and hyperparameters are loaded and then distributed to each device. Subsequently, each device runs the Actor-Critic agent on local environments to gather experience, compute advantages, and update gradients. Finally, gradients are averaged across all devices, and the updated parameters are broadcast to maintain a globally consistent policy.

5.1 Model initialization and data collection

At the start of training, model parameters are replicated across all GPUs, with each device initialized using the same random seed to ensure consistent initial conditions. During training, each GPU interacts with a set of environments to collect data. In single-GPU mode, all environments are handled by one device, whereas in multi-GPU mode, environments are evenly distributed across multiple GPUs.

In single-GPU mode, the agent interacts with N_{envs} environments, collecting transitions over T time steps. At each time step, the policy network computes actions based on the current observations, and the environment transitions accordingly. Rewards, episode completions (done signals), and other relevant information are recorded at each step.

In multi-GPU mode, the total number of environments is divided equally among the available GPUs. For example, with $N_{\text{envs}} = 768$ environments and $N_{\text{devices}} = 3$ GPUs, each device manages 256 environments. Each GPU interacts independently with its assigned environments, collecting

Table 1: Hyperparameters used in the training process.

Hyperparameter	Value	Description
Learning Rate	3×10^{-4}	Optimizer's lr of adam.
Environments per Device	256	Number of envs per device.
Number of Steps	2048	Steps per update.
Total Timesteps	5×10^{8}	Total training timesteps.
Update Epochs	10	Epochs per update.
Number of Minibatches	64	Minibatches per update.
Discount Factor	1	Future reward discount factor.
GAE Lambda	0.95	Bias-variance trade-off.
Clip Epsilon	0.2	PPO clipping parameter.
Entropy Coefficient	0.00	Entropy regularization coef.
Value Function Coefficient	0.5	Value function loss coefficient.
Max Gradient Norm	0.5	Gradient clipping norm.
Activation Function	tanh	Neural network activation.
Learning Rate Annealing	False	Learning rate annealing.
Environment Normalization	True	Environment normalization.

transitions over T time steps and recording the resulting transitions and rewards. This decentralized data collection increases sample diversity since each GPU explores different environments in parallel.

5.2 Minibatch Creation and Model Updates

After data collection, the transitions are reshaped and shuffled to create minibatches for training. The batch size is determined by multiplying the number of environments by the number of time steps, ensuring uniformity across different modes of operation.

In single-GPU mode, the entire dataset is processed on one device. The Adam optimizer is used to update the model parameters, with gradients computed from the policy loss, value function loss, and entropy regularization term. The combined loss function incorporates these components, and the resulting gradients are directly applied to update the model parameters.

In the multi-GPU mode utilizes JAX's `pmap` function to parallelize the training across devices. Each GPU computes its local gradients based on the minibatches created from its assigned data. These local gradients are then averaged across all GPUs using `jax.lax.pmean` to ensure synchronized updates. Specifically, if g_i represents the gradients computed on GPU i, the averaged gradient across N_{devices} GPUs is computed as:

$$g_{\text{avg}} = \frac{1}{N_{\text{devices}}} \sum_{i=1}^{N_{\text{devices}}} g_i.$$

The averaged gradient is then used by each GPU to update its model parameters, ensuring consistency across devices after every update.

$$N_{\text{envs per device}} = \frac{N_{\text{envs}}}{N_{\text{devices}}} = 256 \text{ environments.}$$

Each GPU interacts with its assigned environments independently, collecting data in parallel. Despite operating independently, the GPUs follow the same procedure: they collect transitions over T time steps with parallel environment, compute actions using their local policy networks, and record the resulting transitions and rewards. This decentralized data collection allows for increased sample diversity, as each GPU gathers data from different environments.

In both single-GPU and multi-GPU modes, the loss function remains consistent, combining the policy loss, value function loss, and entropy regularization. The difference lies in the way gradients are computed and applied. In the single-GPU mode, gradients are computed using the entire dataset, with no need for synchronization between devices. Conversely, in the multi-GPU mode, gradients

are computed locally on each device, and synchronization is achieved through gradient averaging, ensuring that all devices maintain consistent model parameters after each update. In single-GPU mode, no synchronization is needed. Conversely, in multi-GPU mode, local gradients are averaged via an all-reduce operation, and the resulting global gradient is used to update parameters on each device.

5.3 Training efficiency and communication overhead

The single-GPU mode offers a simpler training procedure, as all computations are performed on a single device, and no inter-device communication is required. This simplicity, however, comes at the cost of limited computational capacity, as the training process is constrained by the memory and processing power of the single GPU. In contrast, the multi-GPU mode enables parallel processing, allowing for faster training and the ability to handle larger datasets. However, the benefits of multi-GPU training are tempered by the communication overhead introduced by gradient synchronization. The need to average gradients across devices can introduce delays, potentially offsetting some of the gains from parallelization.

Despite these challenges, the multi-GPU mode remains advantageous for large-scale training tasks, where the ability to distribute data and computations across multiple devices outweighs the cost of inter-device communication. The synchronized updates ensure that the model parameters remain consistent across all devices, preserving the integrity of the training process.

5.4 Implementation Details and Communication Overhead

While multi-GPU training accelerates data collection and gradient computation, the synchronization step introduces additional overhead. Specifically, each gradient-update cycle requires inter-device communication to average gradients before parameters can be updated. In practice, for larger batch sizes, the overall throughput gain typically outweighs the cost of communication, as the parallel sampling and parallel gradient calculations significantly reduce the total wall-clock time.

6 Evaluation

6.1 Performance Comparison with Baseline Methods

The performance of the RLA was evaluated using several types of QUBO instances, including formulations derived from the TSP and the Quadratic Assignment Problem (QAP). The TSP instances originated from TSPLIB [22], while the QAP instances were obtained from QAPLIB [23]; both were subsequently transformed into QUBO form by appropriately encoding their cost structures. In addition, random QUBO instances were generated from uniform random numbers ranging from -1000 to 1000, including 0. The well-known n-Queen puzzle was likewise cast into a QUBO formulation, allowing each placement constraint to be captured by the QUBO coefficients. Finally, the k2000 instance was introduced as a benchmark for the maximum cut problem on a complete graph of 2000 vertices, where each edge weight was randomly drawn from the set $\{-1, +1\}$ [24]. All experiments were conducted on a system with the following hardware specifications: CPU: AMD EPYC 7502P (32 cores/64 threads, clocked at 2.5 GHz), 3 NVIDIA RTX A6000 GPUs, 128 GB of memory, and Ubuntu 20.04 LTS as the operating system.

In this study, we set up the PositiveMin-Min Algorithm and three patterns of temperature schedules for Max-Min Algorithm and conducted verification for each algorithm. Table 2 shows the details of the temperature schedules and with each algorithm being computed for 5×10^8 iterations, similar to max-step, and the best solution was recorded.

Table 3 presents the best solutions obtained by each algorithm, with the top-performing values highlighted in bold. Table 4 and Fig. 4 show the optimality gap (optgap) from the global optimal target. An asterisk (*) indicates that the algorithm did not converge to a local optimum even at the maximum number of steps. In such cases, the minimum value across all steps is provided for reference, and these results are omitted from the graphical representation in Fig. 4.

Table 2: Temperature scheduling for the TA algorithms.

Name	Change Frequency	Update Equation
Max-Min-1	1 step	$T_{new} = T_{old} - 1/\text{maxstep}$
Max-Min-2	100 steps	$T_{new} = 0.99 \times T_{old}$
Max-Min-3	10000 steps	$T_{new} = 0.5 \times T_{old}$

Table 3: Energy values of the best solutions found by different algorithms across various QUBO instances. The best values are highlighted in bold. An asterisk (*) indicates that the algorithm did not converge to a local optimum even at the maximum number of steps.

Instance	Type	Matrix Size	Target	Single-RLA	Multi-RLA	PositiveMin-Min	Max-Min-1	Max-Min-2	Max-Min-3
gr24	tsp	529	−21728	−21007	**−21578**	−20915	−20808	−20881	−20997
gr48	tsp	2209	−41954	**−39563**	−39091	−35933	−38517	−38497	−37923
berlin52	tsp	2601	−94458	−88546	**−89774**	−82163	−85339	−87706	−86722
pr76	tsp	5625	−1391841	−1270354	**−1276102**	−1136677	−1254529	−1265690	−1225307
r1k	random	1024	−5620378	−5618528	**−5620378**	−5620240	**−5620378**	**−5620378**	**−5620378**
r2k	random	2048	−15980143	**−15980143**	**−15980143**	−15977009	**−15980143**	**−15980143**	−15980102
nug30	qap	900	−23876	−22800	**−23122**	−22742	−22678	−22734	−22786
k2000	maxcut	2000	−33337	−33214	**−33335**	−32278	−33273	−32139	−32381
nqueen100	nqueen	10000	−100	**−100**	**−100**	−39*	−96	−96	−97

As shown in Table 5, Multi-RLA demonstrates superior performance in discovering the best solutions compared to Single-RLA. However, the standard deviation of the solutions found by Multi-RLA tends to be larger than that of Single-RLA. This suggests that while Single-RLA tends to overfit to local optima, Multi-RLA, which updates the model differently, exhibits greater diversity in its actions by leveraging a broader range of experiences. This increased diversity allows Multi-RLA to explore more globally optimal solutions.

In terms of execution time, as shown in Table 5 and illustrated in Figure 5, the execution time increases linearly with matrix size. This is likely due to the computational complexity of the `CALCULATE_Delta` function, which scales as $\mathcal{O}(N)$. While there is some overhead from communication between GPUs during model updates, the results indicate that in Multi-RLA mode, the execution time decreases linearly with the number of GPUs, demonstrating efficient parallelization.

One key reason that Multi-RLA outperforms Single-RLA is its ability to leverage parallel sampling across multiple GPUs, which leads to a more substantial volume of experience being collected at each training iteration. Because each GPU can simultaneously interact with the environment and produce distinct trajectories, the model effectively sees a larger and more diverse batch of data before each update. As a result, gradient estimates become more stable, and the agent avoids premature convergence on local optima. Furthermore, the presence of multiple random seeds across GPUs naturally increases the variety of trajectories, helping the model to explore a broader set of possible states and actions. This heightened diversity in the training data not only reduces the risk of overfitting but also facilitates discovery of solutions with higher global optimality.

In addition, the computational load of environment interactions and neural network forward–backward passes is distributed across multiple devices in RLA-Multi. By dividing the workload in this way, the algorithm shortens the wall-clock time needed to process a given number of steps, which allows for more extensive experimentation within the same time budget. Although there is communication overhead when aggregating gradients through all-reduce and synchronizing parameters, the net effect is still a speedup overall, because parallel sampling and faster batch processing outweigh the cost of inter-GPU communication. This practical benefit is evident in the linear decrease in total execution time relative to the number of GPUs, as highlighted in Table 5 and Figure 5. Consequently, RLA-Multi not only achieves better exploration but also maintains a more efficient training cycle, which explains why it attains superior performance in discovering high-quality solutions.

Table 4: Optgap percentages for different algorithms across various QUBO instances. An asterisk (*) indicates that the algorithm did not converge to a local optimum even at the maximum number of steps.

Instance	Single-RLA (%)	Multi-RLA (%)	PositiveMin-Min (%)	Max-Min-1 (%)	Max-Min-2 (%)	Max-Min-3 (%)
gr24	3.32	0.690	3.74	4.24	3.90	3.37
gr48	5.70	6.82	14.4	8.19	8.24	9.61
berlin52	6.26	4.96	13.0	9.65	7.15	8.19
pr76	8.74	8.32	18.3	9.87	9.07	12.0
r1k	0.0329	0	0.00246	0	0	0
r2k	0	0	0.0196	0	0	0
nug30	4.51	3.16	4.75	5.01	4.79	4.57
k2000	0.369	0.00560	3.18	0.192	3.59	2.87
nqueen100	0	0	61*	4	4	3

Table 5: Performance comparison of RLA Single and Multi across various QUBO instances.

Single-RLA

Instance	Best	Std	execution time [s]
gr24	−21007	163.31	1277.51
gr48	−39563	222.86	1815.16
berlin52	−88546	491.97	1746.62
pr76	−1270354	5671.35	2574.79
r1k	−5618528	4320.86	1520.99
r2k	−15980143	980.22	1747.64
nug30	−22800	107.97	1323.48
k2000	−33214	95.83	1588.97
nqueen100	−100	0	3986.86

Multi-RLA

Instance	Best	Std	execution time [s]
gr24	−21578	1537.34	493.44
gr48	−39091	318.56	705.71
berlin52	−89774	1279.37	726.10
pr76	−1276102	12736.07	975.17
r1k	−5620378	7280.39	569.67
r2k	−15980143	20671.80	654.67
nug30	−23122	1644.21	574.31
k2000	−33335	635.13	585.00
nqueen100	−100	0.288	1421.93

6.2 Ablation Study on Input Features

We conducted a single feature drop ablation study to assess the individual contributions of each of the five features. Specifically, we created five experimental settings, each excluding exactly one feature: "Std-Off," "Mean-Off," "IQR-Off," "POS-DELTA-Off," and "POS-VEC-Off." These settings maintained identical learning and search conditions to the original Multi-RLA configuration, varying only the availability of the respective features.

The results of these single feature ablation experiments (Tables 6 and 7) clearly illustrate the impact of removing each individual feature. Excluding either of the sign features related to POS (POS-DELTA, POS-VEC) consistently results in performance degradation in various instances of problems, such as gr24 (−0.36%, −0.69%) and k2000 (−0.44%, −0.51%), although this effect is less evident in saturated instances (e.g., r1k, r2k and nqueen100). The Mean feature generally contributes positively to performance, with only marginal improvements observed upon its exclusion (e.g., gr48: +0.12%). Regarding the two dispersion-related features (Std and IQR), their contributions are complementary, and their relative usefulness varies depending on instance scale and structure. For smaller to medium-sized TSP instances, excluding IQR (IQR-Off) significantly enhances performance (gr48: +2.67%, berlin52: +0.50%), whereas for the larger instance pr76, excluding Std (Std-Off) yields superior results (+0.23%). However, for critical instances such as gr24 (TSP), nug30 (QAP), and k2000 (Max-Cut), the full feature set in Multi-RLA consistently provides the best performance, outperforming all single-feature removal configurations. These observations collectively highlight the complementary nature of sign (POS), location (Mean), and dispersion (Std/IQR) statistics. Consequently, we infer that retaining all five features may enhance robustness across a wide variety of problem instances.

Table 6: Single-Feature Drop Ablation Study. Each column removes exactly one feature from the five-feature Multi-RLA while keeping training and search settings identical to the baseline. Entries are QUBO objective values (lower is better); **bold** marks the best value per instance. Target denotes the optimum or best-known reference.

Instance	Target	Multi-RLA	Std-Off	Mean-Off	IQR-Off	POS-DELTA-Off	POS-VEC-Off
gr24	−21728	**−21578**	−21551	−21517	−21520	−21500	−21430
gr48	−41954	−39091	−38891	−39137	**−40133**	−38617	−38624
berlin52	−94458	−89774	−89840	−89552	**−90227**	−89217	−89343
pr76	−1391841	−1276102	**−1279043**	−1272266	−1271114	−1270590	−1259686
r1k	−5620378	**−5620378**	**−5620378**	**−5620378**	**−5620378**	**−5620378**	**−5620378**
r2k	−15980143	**−15980143**	**−15980143**	**−15980143**	**−15980143**	**−15980143**	**−15980143**
nug30	−23876	**−23122**	−23022	−23092	−23050	−23056	−23090
k2000	−33337	**−33335**	−33175	−33180	−33201	−33187	−33164
nqueen100	−100	**−100**	**−100**	**−100**	**−100**	**−100**	**−100**

Table 7: Percentage change vs Multi-RLA for each feature-selection variant. $\Delta[\%] = (\text{Multi-RLA} - \text{Variant})/|\text{Multi-RLA}| \times 100$. Positive means lower energy (better) than Multi-RLA.

Instance	Δ Std-Off [%]	Δ Mean-Off [%]	Δ IQR-Off [%]	Δ POS-DELTA-Off [%]	Δ POS-VEC-Off [%]
gr24	-0.13%	-0.28%	-0.27%	-0.36%	-0.69%
gr48	-0.51%	+0.12%	+2.67%	-1.21%	-1.19%
berlin52	+0.07%	-0.25%	+0.50%	-0.62%	-0.48%
pr76	+0.23%	-0.30%	-0.39%	-0.43%	-1.29%
r1k	0.00%	0.00%	0.00%	0.00%	0.00%
r2k	0.00%	0.00%	0.00%	0.00%	0.00%
nug30	-0.43%	-0.13%	-0.31%	-0.29%	-0.14%
k2000	-0.48%	-0.46%	-0.40%	-0.44%	-0.51%
nqueen100	0.00%	0.00%	0.00%	0.00%	0.00%

7 Conclusions

In this study, we have proposed a novel RL-based approach of annealing computation for solving the QUBO, called RL-driven annealing (RLA). A QUBO solver is a powerful framework capable of uniformly handling various NP-hard combinatorial optimization problems. Unlike conventional heuristic local search approaches based on Δ-based flip policy, RLA trains the flip policy using deep reinforcement learning. Specifically, RLA normalizes Δ at each step and extracts it as a statistic, enabling autonomous learning of the flip policy for the given QUBO problem. Moreover, by modeling state representations and action spaces as fixed dimensions with continuous values regardless of problem size, we show that our method can acquire a unified flip policy suitable for various QUBO matrix sizes and problem types without explicitly providing Δ.

We implemented a QUBO solver based on RLA on a multi-GPU system so that it effectively generates solutions in parallel. Our experimental results show that an appropriate flip policy can be acquired without explicitly specializing in the problem type. Future work should improve the solution accuracy. Furthermore, we can consider other strategies described in Introduction, where we train a model for several QUBO instances and obtain a more general model for inference.

Acknowledgment

This work was partially supported by the JSPS KAKENHI Grant Number JP24K20754, the JST SPRING Grant Number JPMJSP2110, and the JST Advanced Low Carbon Technology Research

and Development Program (ALCA-Next) under Grant Number JPMJAN24F3.

References

[1] H. Goto, K. Tatsumura, and A. R. Dixon, "Combinatorial optimization by simulating adiabatic bifurcations in nonlinear hamiltonian systems," *Science advances*, vol. 5, no. 4, 2019.

[2] R. Yasudo, K. Nakano, Y. Ito, R. Katsuki, Y. Tabata, T. Yazane, and K. Hamano, "GPU-accelerated scalable solver with bit permutated cyclic-min algorithm for quadratic unconstrained binary optimization," *Journal of Parallel and Distributed Computing*, vol. 167, pp. 109–122, 2022.

[3] K. Tatsumura, A. R. Dixon, and H. Goto, "FPGA-based simulated bifurcation machine," in *International Conference on Field Programmable Logic and Applications (FPL)*, pp. 59–66, 2019.

[4] H. Kagawa, Y. Ito, K. Nakano, R. Yasudo, Y. Kawamata, R. Katsuki, Y. Tabata, T. Yazane, and K. Hamano, "High-throughput FPGA implementation for quadratic unconstrained binary optimization," *Concurrency and Computation: Practice and Experience*, vol. 35, no. 14, p. e6565, 2023.

[5] Y. Su, T. T.-H. Kim, and B. Kim, "Flexspin: A scalable CMOS Ising machine with 256 flexible spin processing elements for solving complex combinatorial optimization problems," in *2022 IEEE International Solid-State Circuits Conference (ISSCC)*, vol. 65, pp. 1–3, IEEE, 2022.

[6] K. Kawamura, J. Yu, D. Okonogi, S. Jimbo, G. Inoue, A. Hyodo, A. L. Garcia-Arias, K. Ando, B. H. Fukushima-Kimura, R. Yasudo, T. V. Chu, and M. Motomura, "Amorphica: 4-replica 512 fully connected spin 336MHz metamorphic annealer with programmable optimization strategy and compressed-spin-transfer multi-chip extension," in *International Solid-State Circuits Conference (ISSCC)*, 2023.

[7] Y.-C. Chu, Y.-C. Lin, Y.-C. Lo, and C.-H. Yang, "30.4 a fully integrated annealing processor for large-scale autonomous navigation optimization," in *2024 IEEE International Solid-State Circuits Conference (ISSCC)*, vol. 67, pp. 488–490, IEEE, 2024.

[8] H.-W. Chiang, C.-F. Nien, H.-Y. Cheng, and K.-P. Huang, "ReAIM: A ReRAM-based adaptive Ising machine for solving combinatorial optimization problems," in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pp. 58–72, IEEE, 2024.

[9] T. Imanaga, K. Nakano, R. Yasudo, Y. Ito, Y. Kawamata, R. Katsuki, S. Ozaki, T. Yazane, and K. Hamano, "Solving the sparse QUBO on multiple GPUs for simulating a quantum annealer," in *2021 Ninth International Symposium on Computing and Networking (CANDAR)*, pp. 19–28, IEEE, 2021.

[10] A. Lucas, "Ising formulations of many NP problems," *Frontiers in Physics*, vol. 2, p. 5, 2014.

[11] R. S. Sutton and A. G. Barto, "Reinforcement learning: an introduction, 2nd edn. adaptive computation and machine learning," 2018.

[12] A. Fawzi, M. Balog, A. Huang, T. Hubert, B. Romera-Paredes, M. Barekatain, A. Novikov, F. J. R. Ruiz, J. Schrittwieser, G. Swirszcz, D. Silver, D. Hassabis, and P. Kohli, "Discovering faster matrix multiplication algorithms with reinforcement learning," *Nature*, vol. 610, no. 7930, pp. 47–53, 2022.

[13] Q. Ma, S. Ge, D. He, D. Thaker, and I. Drori, "Combinatorial optimization by graph pointer networks and hierarchical reinforcement learning," *arXiv preprint arXiv:1911.04936*, 2019.

[14] T. Barrett, W. Clements, J. Foerster, and A. Lvovsky, "Exploratory combinatorial optimization with reinforcement learning," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 34, pp. 3243–3250, 2020.

[15] M. Nazari, A. Oroojlooy, L. Snyder, and M. Takác, "Reinforcement learning for solving the vehicle routing problem," *Advances in neural information processing systems*, vol. 31, 2018.

[16] R. Yasudo, "Bandit-based variable fixing for binary optimization on GPU parallel computing," in *2023 31st Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pp. 154–158, IEEE, 2023.

[17] L. Engstrom, A. Ilyas, S. Santurkar, D. Tsipras, F. Janoos, L. Rudolph, and A. Madry, "Implementation matters in deep policy gradients: A case study on PPO and TRPO," *CoRR*, vol. abs/2005.12729, 2020.

[18] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, pp. 529–533, Feb. 2015.

[19] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang, "JAX: composable transformations of Python+NumPy programs," 2018.

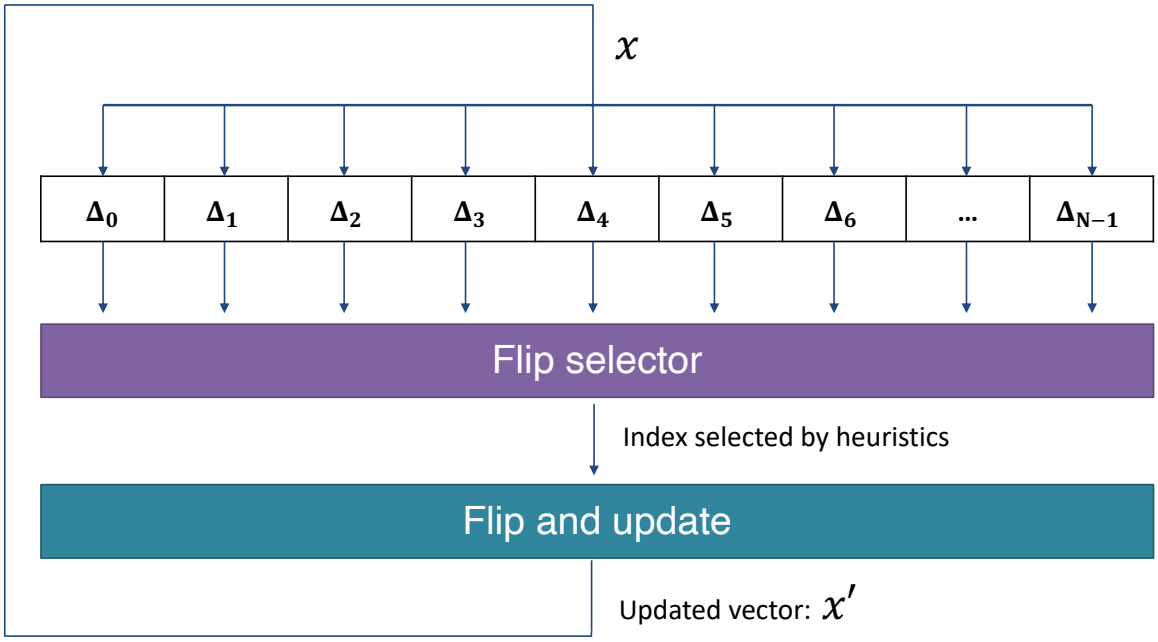
[20] C. Lu, J. Kuba, A. Letcher, L. Metz, C. Schroeder de Witt, and J. Foerster, "Discovered policy optimisation," *Advances in Neural Information Processing Systems*, vol. 35, pp. 16455–16468, 2022.

[21] R. T. Lange, "gymnax: A JAX-based reinforcement learning environment library," 2022.

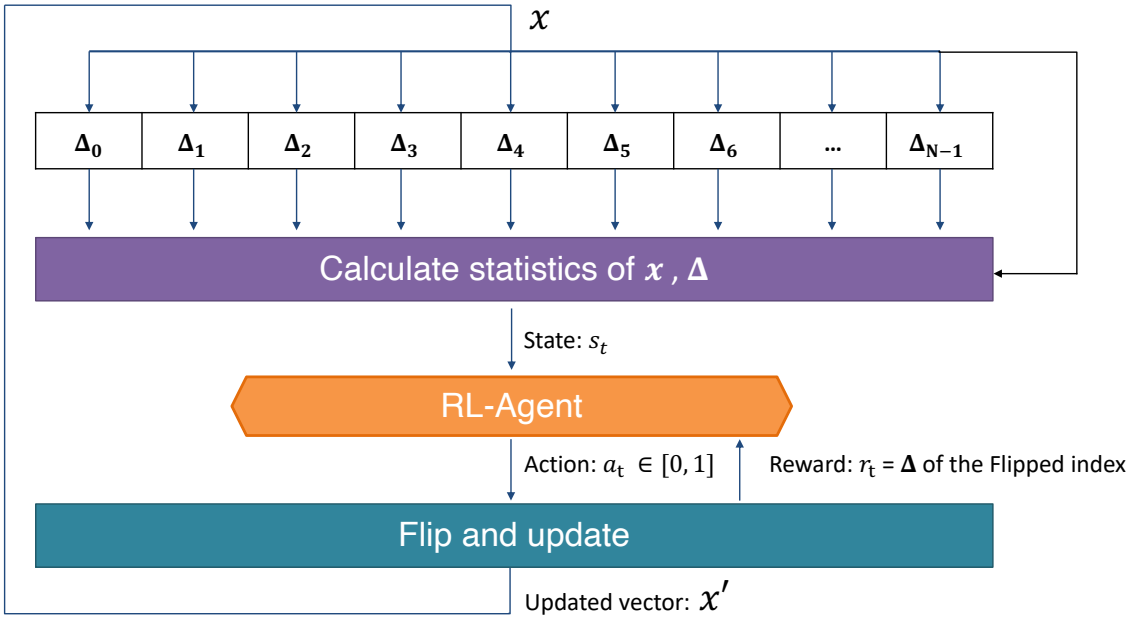
[22] G. Reinelt, "Tsplib: A traveling salesman problem library," *ORSA journal on computing*, vol. 3, no. 4, pp. 376–384, 1991.

[23] "Qaplib." `https://coral.ise.lehigh.edu/data-sets/qaplib/`. Accessed: 2024/01/16.

[24] K. Katsuki, D. Shin, N. Onizawa, and T. Hanyu, "Fast solving complete 2000-node optimization using stochastic-computing simulated annealing," in *2022 29th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pp. 1–4, 2022.



(a)



(b)

Figure 1: Annealing computation based on the Δ-based flip policy for the N-dimensional state vector $\boldsymbol{x}$. (a) Conventional method. (b) Proposed RL-driven method.

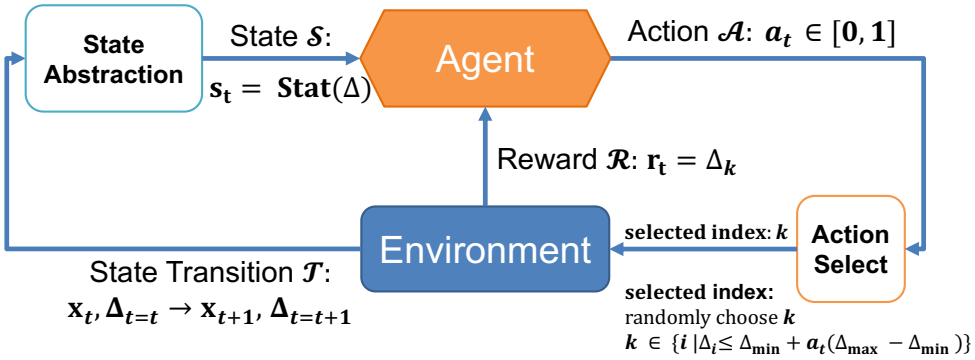


Figure 2: Overview of the RL Framework in RLA.

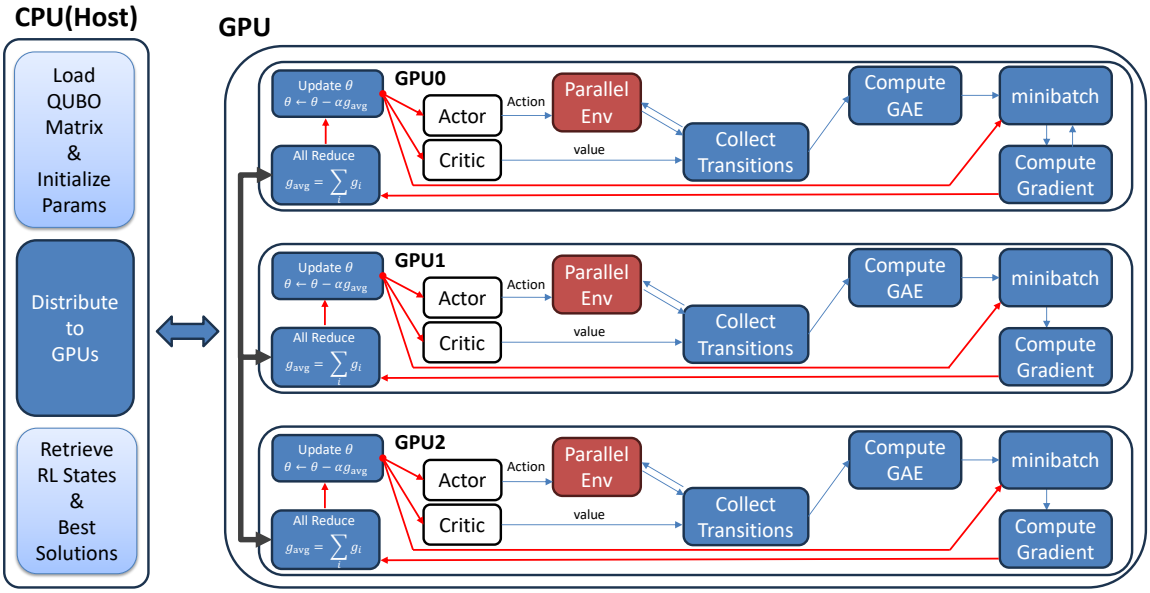


Figure 3: Overview of Multi-RLA workflow with three GPUs, each of which contains 256 RL environments.

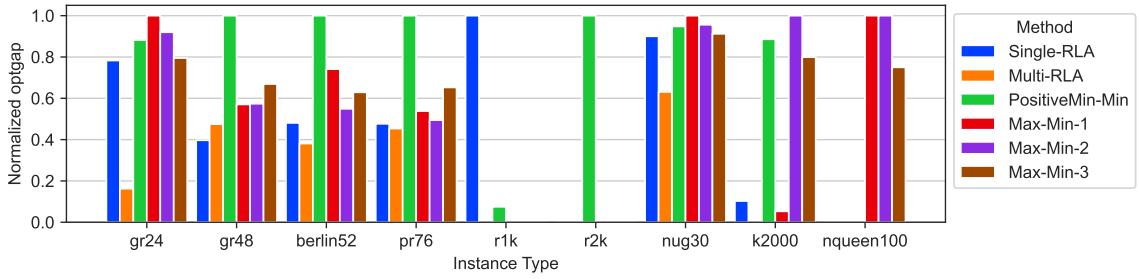


Figure 4: Optimality gap values for different algorithms across various QUBO instances. The values are normalized by the worst value for each problem instance. Note that PositiveMin-Min for nqueen100 is omitted because it is an outlier.

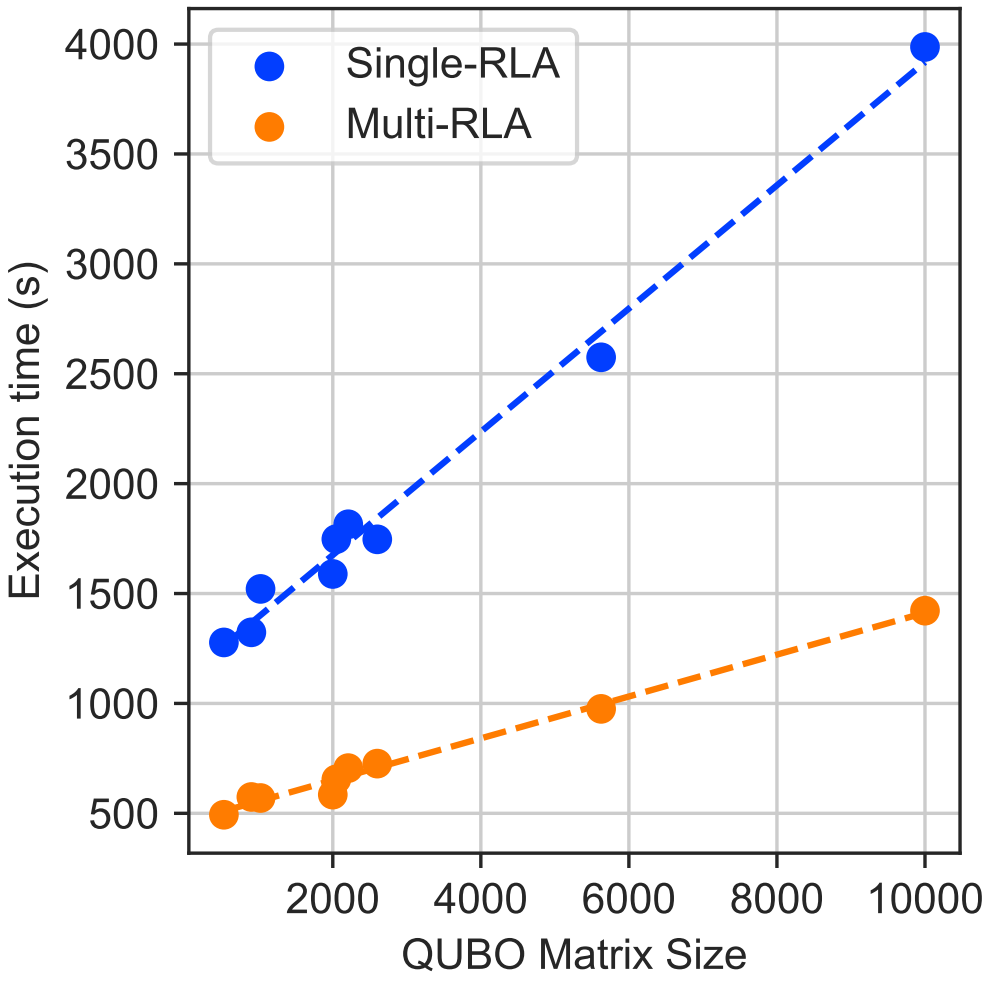


Figure 5: The change of execution time of Single-RLA and Multi-RLA with the size of QUBO instances.