RLTuner: Floating-Point Precision Tuning Using Deep Reinforcement Learning

Haneul Lee*
Department of Computer Science
University of California, Davis
Davis, USA
hnee@ucdavis.edu

Yutong Wang*
Department of Computer Science
University of California, Davis
Davis, USA
ytwwang@ucdavis.edu

Cindy Rubio-González
Department of Computer Science
University of California, Davis
Davis, USA
crubio@ucdavis.edu

***Abstract*—Floating-point precision tuning is essential for optimizing numerical software, involving the selection of the ideal precision for floating-point variables to balance performance and accuracy. Identifying the optimal configuration is challenging due to the large search space and the interactions between precision and speed. Existing works get stuck in local optima, removing a variable and/or a function call from the search space immediately if it fails to meet a given error threshold. We introduce RLTuner, a novel automated tuning framework that reformulates precision tuning as a Markov Decision Process (MDP) and employs a Deep Q-Network (DQN) to identify superior configurations that traditional pruning-based methods overlook. We evaluate RLTuner on 20 numerical benchmarks, ranging from kernels to large-scale applications like LULESH. RLTuner outperforms state-of-the-art baselines in 17 out of 20 cases, achieving speedups of up to 46.28%, and yields the lowest error in 11 out of 20 cases with up to 10 orders of magnitude higher accuracy. Notably, it identifies higher-quality configurations—yielding both superior performance and lower error—in 40% of benchmarks.**

***Index Terms*—floating-point arithmetic, dynamic analysis, precision tuning, reinforcement learning, DQN**

I. Introduction

With the growing reliance on numerical software in diverse computational domains, achieving a balance between performance and accuracy has become a critical research focus [1]–[8]. High performance is often achieved through program optimization techniques that improve computational efficiency in numerical software, but this can lead to the cost of increased computational error, especially when working with floating-point arithmetic. While higher precision ensures lower error and greater reliability, it may significantly hinder performance and lead to unnecessary resource consumption. Finding a balance between speed and accuracy is key to designing numerical applications, highlighting the need for automated techniques that optimize performance while reducing numerical error.

In the past decade, floating-point precision tuning has emerged as a promising program optimization technique to explore the trade-off between performance and accuracy in numerical software [1], [6]. Given a numerical program, representative inputs and correctness criteria, floating-point precision tuners search the space of program floating-point variables, available floating-point precisions (e.g., `double` and `float`), and calls to mathematical functions for which a lower-precision implementation is available to identify one or more "optimal" mixed-precision program variants that maximize performance while ensuring accuracy. In this process, precision tuners explore precision configurations [9] that specify the floating-point types assigned to specific variables, and/or alternative mathematical functions to call. Precision tuning is non-trivial: exploring the full search space is infeasible even for small programs, and blindly assigning floating-point precision can negatively impact both accuracy and performance due to type castings and missed opportunities for data vectorization.

To tackle these challenges, various techniques have been proposed for automated precision tuning. Dynamic search-based approaches [2]–[5], [9]–[13] are the most popular for tuning numerical programs that have been utilized for scientific computing workloads, which have been summarized in the literature [1], [6]. These approaches use greedy algorithms to explore the configuration space of a given target program by executing mixed-precision variants to evaluate their performance and correctness. Among various dynamic approaches, a delta-debugging-inspired algorithm introduced in Precimonious [9] is the most commonly used, either as a baseline or as a framework component [2], [4], [5], [7], [10], [11], [13]. It has been shown to outperform other search algorithms in terms of the quality of the resulting variants [4], [11], [14]. Tools like Precimonious aim to identify optimal precision configurations that balance high performance with sufficient accuracy. While these tools can save time and provide strong results, discontinuities in a program may mislead greedy algorithms into local optima that are far from the global optimum [6]. As a result, making significant improvements or breakthroughs in precision tuning quality remains a challenge. This challenge calls for more advanced approaches to precision tuning that can efficiently navigate the search space.

Reinforcement Learning (RL) is a powerful machine-learning technique inspired by behavioral psychology, where an agent learns to make decisions through interactions with an environment [16], [17]. RL enables efficient exploration of large search spaces by associating actions with outcomes based on a reward function. The agent iteratively refines its

*Both authors contributed equally.

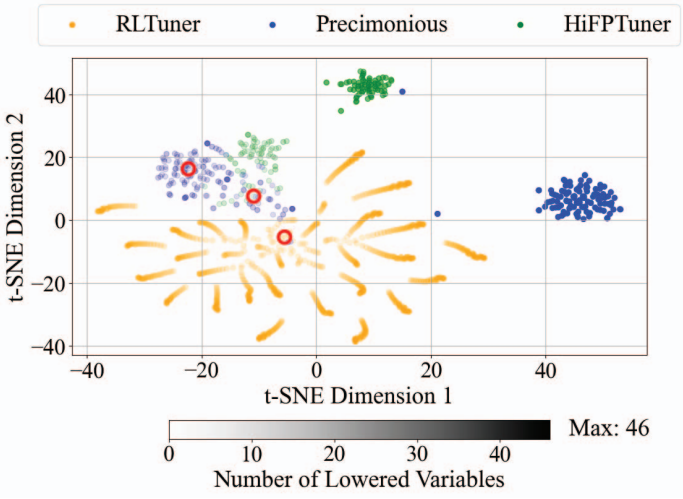


Fig. 1: t-SNE plot for LavaMD from Rodinia [15]

strategy, or policy, through trial-and-error learning, aiming to maximize cumulative rewards. RL has been widely applied in various domains, including program optimization, as demonstrated in prior research [18]–[21]. However, its potential for floating-point precision tuning remains unexplored. Floating-point precision tuning naturally exhibits characteristics that make it well suited to RL. The configuration space is large and combinatorial, the effects of precision choices are non-linear and program-dependent, and the performance–accuracy trade-off can only be evaluated through program execution. These characteristics make greedy and heuristic search strategies prone to suboptimal solutions.

In this work, we propose to navigate the high-dimensional and non-linear search space of precision configurations using Deep Reinforcement Learning (DRL), specifically employing a Deep Q-Network (DQN). Unlike traditional greedy heuristics that are prone to local optima, a DQN-based approach leverages deep neural networks as function approximators to navigate the precision search spaces. This allows the agent to identify collective variable groupings that yield significant speedups, even when individual changes provide no immediate gain, i.e., a scenario where greedy search typically fails. By balancing exploration and exploitation, RLTuner ensures a more robust and comprehensive search for optimal configurations.

To the best of our knowledge, this is the first work to formulate floating-point precision tuning as a DRL problem. We model the tuning task as a Markov Decision Process (MDP) and implement our approach in a tool called RLTuner. By leveraging the DQN algorithm, RLTuner employs a systematic trial-and-error mechanism to identify optimal precision configurations that remain hidden to traditional search strategies. Specifically, our framework uses deep neural networks to approximate the value of complex configuration states, enabling a principled exploration of the high-dimensional trade-off space between performance and accuracy.

We evaluate RLTuner on 20 representative benchmarks across various domains and compare its effectiveness against two state-of-the-art baselines: Precimonious [9] and HiFPTuner [5]. Our experimental results show that RLTuner identifies precision configurations that outperform the baselines in 85% (17 out of 20) of the benchmarks, achieving speedups of up to 46.28%. Beyond performance, RLTuner yields the lowest relative error in 55% (11 out of 20) of the cases, with some configurations achieving up to 10 orders of magnitude higher accuracy than existing approaches. Notably, RLTuner-optimized programs achieve both the highest speedup and the lowest relative error in 40% (8 out of 20) of benchmarks, demonstrating its ability to find higher-quality configurations that traditional greedy heuristics overlook. On the largest benchmark, LULESH, RLTuner identifies a superior configuration with 17.89% speedup, reducing the search time by 33% compared to Precimonious. Furthermore, RLTuner exhibits strong search robustness, maintaining a performance lead across varying error thresholds from 10^{-4} to 10^{-8}.

In summary, our paper makes the following contributions:

- The first formulation of floating-point precision tuning as a DRL task. We model the search space as a Markov Decision Process to enable principled exploration (Section III).
- The design and implementation of RLTuner, the first DQN-based tool for automatically tuning the precision of floating-point variables and mathematical functions in C/C++ programs (Section IV).
- A comprehensive evaluation on 20 benchmarks shows RLTuner outperforms baselines in 17 cases, achieving up to 46.28% speedup. We demonstrate that RLTuner identifies higher-quality configurations—yielding both better performance and significantly higher accuracy (up to $10^{10}\times$)— by effectively navigating the precision-performance trade-off (Section IV).

II. Motivating Example and Overview

A. Motivating Example

This section demonstrates the need for a precision tuning approach that explores the search space more effectively to balance performance gains with numerical stability. To illustrate this, we present a motivating example using the LavaMD benchmark from Rodinia [15], comparing our approach, RLTuner, against two state-of-the-art baselines: Precimonious and HiFPTuner (further details are provided in Section IV-B).

Figure 1 presents a t-SNE that visualizes the high-dimensional configuration space of LavaMD. Each dot in the plot represents a unique configuration/precision assignment, where the spatial proximity between dots reflects their similarity. Increased color solidity indicates a higher proportion of low-precision variables. While red circles highlight the best configurations found, the distribution of the dots reveals a stark contrast in search strategy: RLTuner (orange) achieves broader coverage of the space, whereas Precimonious (blue) and HiFPTuner (green) remain confined to more localized

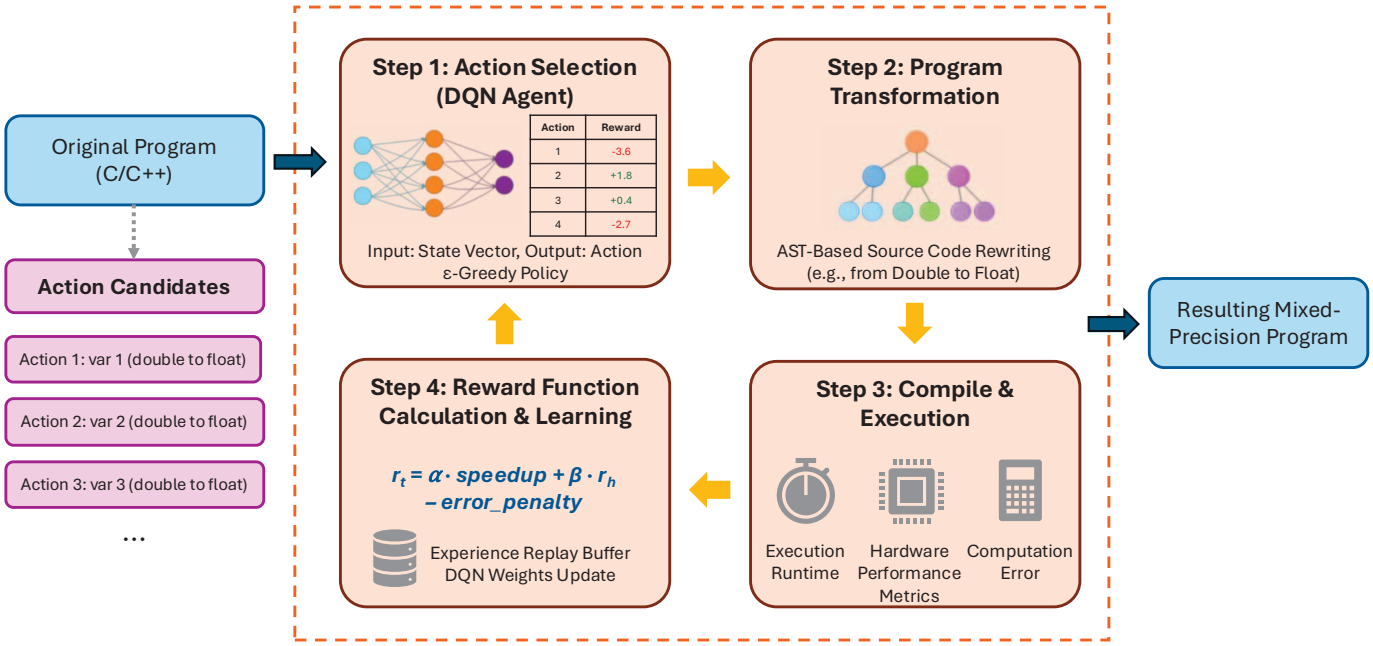


Fig. 2: Overview of Deep Reinforcement Learning-based precision tuning approach

clusters. This clustering is a direct result of their respective reliance on binary-search delta-debugging and hierarchical community structures, demonstrating the limitations of these heuristics when navigating high-dimensional, suboptimal regions.

Precision tuner typically require a user-defined error threshold, which acts as a configurable tolerance level for the maximum allowable deviation from high-precision ground truth. In this example, we adopt a target threshold of 10^{-4}, a permissive limit that intentionally expands the opportunities for optimization. We observe that RLTuner's broader search space coverage enables it to discover superior configurations that baselines fail to locate. Notably, RLTuner identifies a configuration delivering a 13.91% speedup with zero relative error. In contrast, Precimonious achieves only a marginal 3.04% speedup with 1.50E-07 error, while HiFPTuner yields no speedup. These results demonstrate that RLTuner's Deep Reinforcement Learning-based exploration navigates complex search spaces more effectively than traditional heuristics, identifying 'sweet-spot' configurations that maximize the trade-off between performance and accuracy.

B. Overview of Reinforcement Learning and RLTuner

a) Approximation-Based RL and DQN: RL is a machine-learning paradigm where an agent learns to make a sequence of decisions to maximize a cumulative reward through iterative interactions with an environment [16], [17]. RL methods can be categorized into two types: (1) tabular-based and, (2) approximation-based [22]. Traditional RL methods, such as Q-learning, maintain a tabular Q-table to store the expected utility (Q-value) for every state-action pair. However, the search space for mixed-precision tuning is discrete and exponential—specifically L^{n+m}, where L is the number of available precision levels and n, m are the counts of tunable variables and functions. This makes tabular methods computationally intractable due to the curse of dimensionality. To address this, RLTuner employs a DQN which replaces the static Q-table with a Deep Neural Network (DNN) as a high-dimensional function approximator to estimate Q-values, making them ideal for the exponential state spaces inherent in floating-point tuning.

Unlike typical supervised learning that requires extensive offline training, RLTuner treats precision tuning as an online optimization problem. For each target benchmark, the agent initializes with no prior knowledge and refines its policy through active exploration. DQN is an off-policy algorithm, meaning it can learn from experiences generated by previous versions of the tuning policy. This is facilitated by experience replay, where past transitions are stored in a buffer and sampled to update the network. In HPC scenarios where program execution is time-consuming, this off-policy nature is critical: it enables high sample efficiency by reusing historical execution data, allowing the agent to efficiently navigate the non-linear relationship between precision and performance.

b) RLTuner*:* We developed RLTuner, an automated precision tuning framework that employs DQN to optimize floating-point variable types and mathematical functions in C/C++ applications. RLTuner is designed to maximize execution speedup while strictly adhering to user-defined error thresholds. As illustrated in Figure 2, RLTuner orchestrates a feedback-driven optimization loop consisting of four primary components:

(1) *Action Selection.* The DQN agent observes the current precision state s_t and selects an action a_t based on an ϵ-greedy policy. An action corresponds to lowering the precision of a specific tunable candidate: changing the floating-point type of a variable from `double` to `float`, or calling a lower-precision implementation of a mathematical function (e.g.,

Algorithm 1 Mixed-precision search using DQN

Input: original configuration C_{original}
Output: proposed configuration C_{proposed}
Initialize:
Initialize replay buffer $\mathcal{D}$ with capacity N
Initialize Q-network Q with random weights θ
Initialize target network $\hat{Q}$ with $\theta^- \leftarrow \theta$
$s_0 \leftarrow$ initial state; $C_{\text{proposed}} \leftarrow C_{\text{original}}$
for episode $= 1$ to $EP_{\max}$ **do**
  $s \leftarrow s_0$
  $\mathcal{A}_{used} \leftarrow \emptyset$
  for $t = 1$ to $|\mathcal{A}|$ **do**
    Select a_t by ϵ-greedy from $\mathcal{A} \setminus \mathcal{A}_{used}$
    $\mathcal{A}_{used} \leftarrow \mathcal{A}_{used} \cup \{a_t\}$
    Apply a_t, observe s_{t+1}, and calculate reward:
      $r_t = \alpha \cdot \text{speedup} + \beta \cdot r_h - \text{error_penalty}$
    Store (s_t, a_t, r_t, s_{t+1}) in $\mathcal{D}$
    if $|\mathcal{D}| \geq N_{\text{init}}$ **then**
      Sample mini_batch and compute:
        $y_j = r_j + \gamma \max_{a'} \hat{Q}(s_{j+1}, a'; \theta^-)$
      Update θ by minimizing $(y_j - Q(s_j, a_j; \theta))^2$
    end if
    if $t \bmod C = 0$ **then**
      $\theta^- \leftarrow \theta$
    end if
    $s \leftarrow s_{t+1}$
  end for
end for
return C_{proposed}

`sinf` instead of `sin`).

(2) *Program Transformation.* Based on the selected action, RLTUNER utilizes a Clang-based AST transformation engine to rewrite the source code. This component automatically updates type declarations and function calls, ensuring the abstract precision state is accurately reflected in the executable code.

(3) *Compile & Execution.* The transformed program is compiled and executed. Unlike traditional tuners that rely solely on execution time, RLTUNER performs multi-dimensional profiling. We capture not only the wall-clock speedup and numerical relative error but also architectural hardware metrics to provide a richer feedback signal.

(4) *Reward Function Calculation & Learning.* The agent receives a composite reward r_t that incorporates performance gains, hardware efficiency, and a logarithmic error penalty. The reward is used to update the agent's knowledge, influencing its future action choices. This sophisticated reward shaping guides the agent through the high-dimensional search space, prioritizing configurations that maximize the performance speedup while adhering to numerical accuracy constraints.

The loop continues iteratively, allowing the agent to refine its policy through online experience. The process terminates upon reaching a user-defined condition, such as a maximum number of episodes or a target performance threshold. Detailed algorithmic implementation is provided in Section III-A2.

III. Technical Approach

Given a floating-point program, our goal is to identify a mixed-precision configuration that maximizes program speedup while adhering to a user-defined error threshold. As shown in Figure 2, RLTUNER leverages a DQN agent that observes the program's precision state and iteratively adjusts configurations by maximizing a reward signal that penalizes accuracy violations while rewarding performance gains.

A. Precision Tuning as a Reinforcement Learning Problem

1) Markov Decision Process Formulation: We model the problem of floating-point precision tuning as a Markov Decision Process (MDP). An MDP is defined by a tuple (S, A, P, R, γ), where S is the set of states, A is the set of actions, P is the transition function, R is the reward function, and γ is the discount factor. The details of each component are as follows:

(S) A state $s \in S$ represents a specific precision configuration of the target program, represented as a vector $\mathbf{v} = [v_1, v_2, \ldots, v_{n+m}]$. Each element v_i denotes the precision type (e.g., `float` or `double`) of a tunable variable or function call. The initial state s_0 is defined as the configuration where all elements are set to the highest available precision, i.e., `double`. The size of state space is 2^{n+m}, where n and m are the counts of tunable variables and function calls, respectively.

(A) An action $a_t \in A$ is defined as selecting a specific variable or function call to assign lower precision (e.g., changing the type of a floating-point variable from `double` to `float`, or calling a lower-precision implementation of a function instead such as `sinf` instead of `sin`). At each step t, the agent chooses one element from the set of $(n + m)$ candidates to transform.

(P) The transition $P(s_{t+1}|s_t, a_t)$ defines how an action transforms the current state into a new state. Upon choosing an action a_t, the environment updates the precision of the corresponding element in the state vector, resulting in a unique successor state s_{t+1}. This process mimics the systematic exploration of the configuration space, and is deterministic in our framework.

(R) The reward function $R(s_t, a_t)$ evaluates the quality of a precision transition by balancing performance gains and numerical accuracy. We define the reward as:

$$R = \begin{cases} \mathcal{F}_{\text{perf}} - \text{error_penalty}, & \text{if Error}(s_{t+1}) \leq \epsilon_T \\ -P, & \text{otherwise} \end{cases} \quad (1)$$

where ϵ_T is the user-defined error threshold. For valid configurations, the reward consists of a performance term $\mathcal{F}_{\text{perf}}$ (derived from execution speedup and hardware performance metrics) and a fine-grained error penalty, i.e., error_penalty, that encourages maintaining high numerical integrity. If a configuration violates the error threshold or fails to execute, a large constant penalty P is assigned to steer the agent away from invalid regions. This design ensures that the agent prioritizes "precision-safe" configurations while aggressively pursuing execution efficiency.

2) DQN for Precision Tuning: To navigate the high-dimensional and non-linear search space of precision configurations, RLTUNER employs a DQN as a functional approximator. The complete tuning process is detailed in Algorithm 1.

Neural Network Architecture. We implement the Q-network using a Multi-layer Perceptron (MLP). The network consists of an input layer representing the current configuration vector, two fully-connected hidden layers with ReLU activation, and an output layer that predicts Q-values for all $(n+m)$ possible tuning actions. This MLP structure allows the agent to learn the complex interdependencies between program variables and their collective impact on performance.

Search and Exploration Strategy. The algorithm begins by initializing the Q-network with random weights and setting the initial state s_0 to the all-double precision configuration (Algorithm 1, Line 2). To balance the exploration of novel configurations with the exploitation of known high-performance settings, the agent utilizes an ϵ-greedy policy (Line 5). With probability ϵ, the agent selects the action with the highest predicted Q-value; with probability $1-\epsilon$, it explores a random unselected action. To prevent redundant searches and improve efficiency, previously selected actions within the same episode are tracked in $\mathcal{A}_{\text{used}}$ and excluded from the current step's action space.

Reward Shaping with Hardware Performance Metrics. The agent's learning is guided by a composite reward signal r_t computed from the resulting speedup and numerical error:

$$r_t = \alpha \cdot \textit{speedup} + \beta \cdot r_h - \textit{error_penalty} \tag{2}$$

where α and β are weighting factors. To ensure a high sensitivity to numerical precision, RLTUNER defines *error_penalty* as the logarithm of the relative error, which provides a continuous gradient for the agent to optimize. We also incorporate hardware performance metrics as a shaping reward r_h to assist the learning process of RLTUNER. Specifically, we gather data on floating-point operations (FLOPs), LLC cache miss rates, and DRAM memory traffic. Unlike execution time, which can be noisy, hardware metrics provide immediate architectural feedback on how precision reduction alleviates memory pressure. This ensures a denser reward signal: even if a configuration does not yield an immediate wall-clock speedup, improvements in r_h guide the agent toward promising regions of the search space. Any configuration exceeding the error threshold ϵ_T or failing at runtime incurs a substantial penalty, effectively enforcing the accuracy constraint and guiding the agent back to the feasible region of the search space.

Learning Dynamics. RLTUNER incorporates standard DQN stabilization techniques as detailed in Algorithm 1. Each transition (s_t, a_t, r_t, s_{t+1}) is stored in a replay buffer $\mathcal{D}$ (Line 9). Once the buffer reaches threshold N_{init} (Line 10), mini-batches are sampled to update the Q-network. The target Q-value is computed via Bellman backup using a separate target network $\hat{Q}$ (Line 12), and the main network's weights θ are updated by minimizing the mean squared error between predicted and target Q-values (Line 13). Every C steps, the target network's weights θ^- are synchronized with θ (Line 16). Tuning terminates after EP_{max} episodes (Line 1).

B. Transformation and Validation

Once the DQN agent selects an action, RLTUNER applies a source-to-source transformation that maps the state vector s_t back to source code (e.g., replacing `double` with `float` or `sin` with `sinf`). We implement this using a custom Clang-based plugin that operates on the AST to automate these changes. The modified code is then compiled and executed to compute r_t: numerical error is measured against the double-precision reference, while execution time and hardware counters are captured via Linux `perf`.

IV. Experimental Evaluation

This section evaluates the effectiveness of RLTUNER relative to two state-of-the-art baselines: PRECIMONIOUS and HIFPTUNER. Our analysis is guided by the following Research Questions (RQs):

- **RQ1** How effective is RLTUNER at identifying mixed-precision configurations that improve performance while maintaining low accuracy loss compared to the baselines?
- **RQ2** To what extent does RLTUNER improve the exploration of high-dimensional search spaces compared to PRECIMONIOUS and HIFPTUNER?
- **RQ3** How robust is RLTUNER's performance across varying error thresholds when compared to the baseline approaches?
- **RQ4** How does RLTUNER generalize to different input datasets and problem sizes beyond those used during the tuning process?

A. Experimental Setup

Benchmarks. We evaluate RLTUNER on 20 benchmarks categorized into three tiers of increasing complexity:

- *Numerical Kernels:* To isolate floating-point error propagation from system-level effects, we evaluate basic mathematical kernels including arclength, simpsons, and nine linear algebra kernels from PolyBench [23] (e.g., correlation, gemm, syrk).
- *Proxy Applications:* To test more realistic application structures, we include Blackscholes from PARSEC [24] and four applications from the Rodinia suite [15]: CFD, Hotspot, Kmeans, and LavaMD.
- *HPC Mini-Applications:* To evaluate production-relevant scientific workloads, we include cg from the NAS Parallel Benchmarks [25], HPCCG and miniAMR from the Mantevo suite [26], and LULESH 2.0 from Lawrence Livermore National Laboratory [27].

Following the recommendation in precision tuning literature, we utilize representative inputs for the tuning process. Specifically, for small kernels such as arclength and simpsons, we follow their intrinsic mathematical configurations; for the numerical kernels in Polybench, we adopt the `STANDARD_DATASET` size, except for gemver (`LARGE_DATASET`) and bicg (`EXTRALARGE_DATASET`); for cg, we use the provided input Class A; and for other proxy and large-scale applications such as miniAMR and LULESH, we use the default sizes provided by the benchmark suites to reflect realistic HPC workloads.

TABLE I: The parameters of RLTUNER

Parameter	Value
Q-network type	Multi-layer Perceptron (MLP)
Input dimension	d (configuration vector length)
Hidden layer 1 size	256
Hidden layer 2 size	128
Output dimension	$\|\mathcal{A}\|$ (number of actions)
Activation function	ReLU
Loss function	Mean Squared Error (MSE)
Optimizer	Adam
Learning rate	1×10^{-3}
Discount factor γ	0.95
Exploration policy	ϵ-greedy
Initial ϵ	1.0
Final ϵ	0.1
ϵ decay	Linear
Batch size	64
Replay buffer size	50,000
Initial replay size	500
Target network update	Every 100 steps
Number of episodes	30

Floating-Point Precisions and Compilation Flags. We consider double and single precision for floating-point variables and math functions. Specifically, we target the substitution of 64-bit double-precision functions from the C Standard Math Library—including trigonometric (`sin`, `cos`), exponential (`exp`, `log`), and power functions (`pow`, `sqrt`)—with their 32-bit single-precision counterparts (e.g., `sinf`, `expf`, `sqrtf`). All experiments utilize the default compilation flags provided by the respective benchmark suites: most with `-O2` optimization flag, while CFD, Hotspot, HPCCG and LULESH use `-O3`. To maintain consistency, RLTUNER and all baselines utilize these same benchmark-specific compilation flags.

Baselines. To the best of our knowledge, this work represents the pioneer effort in employing Deep Reinforcement Learning, specifically DQN, for automated floating-point precision tuning. To evaluate its efficacy, we compare our approach against two representative dynamic precision tuners: PRECIMONIOUS [9] and HIFPTUNER [5]. PRECIMONIOUS is the de facto standard baseline in the field [2], [5], [10], [13], [28], leveraging delta debugging [29] to navigate the search space, which has been recognized as an effective search strategy in recent precision tuning studies [11], [14]. HIFPTUNER is selected as it incorporates community search and dependency analysis, representing the state-of-the-art in heuristic-driven precision tuning that effectively handles large-scale programs.

Following the taxonomy in [30], we exclude tools that: (1) target restricted scales or use static methods [8], [10], [11], [31]; (2) are tailored for GPUs [28], [32], [33] or narrow scopes such as math library functions [12]; (3) have implementation constraints requiring manual intervention [34], [35]; or (4) are orthogonal to our baselines (FPLEARNER [2] is a predictive meta-tuner, and Blame Analysis [13] is superseded by HIFPTUNER).

Evaluation Metrics. We measure the effectiveness of RLTUNER and the baselines by comparing Equation (3) the best program speedup they can find, and Equation (4) the corresponding incurred computation error. The program speedup is defined as the relative reduction in runtime achieved by a mixed-precision program, calculated using the formula:

$$\text{Speedup} = \frac{\text{Original Runtime} - \text{Mixed-Precision Runtime}}{\text{Original Runtime}} \tag{3}$$

This metric quantifies the relative reduction in runtime achieved by the mixed-precision program compared to the original implementation. The speedup is calculated by average of ten time executions of the benchmarks, except for LULESH for which we report the average of five executions due to its long runtime reducing performance noise, following prior work (e.g., FPLEARNER [2]).

The second metric is the computation error. HPCCG and Kmeans use the Miss Position Percentage (MPP) to measure if the calculation results are acceptable, i.e., MPP equals to zero. For all other benchmarks except for HPCCG and Kmeans, the metric corresponds to the absolute value of the relative error of the result with respect to the result produced by the original program, shown in the following formula:

$$\text{Error} = \frac{|\text{Result}_{\text{new}} - \text{Result}_{\text{original}}|}{\text{Result}_{\text{original}}} \tag{4}$$

Our ideal goal is to discover a mixed-precision program that has higher speedup and smaller error.

Besides the effectiveness of precision tuners, we also care about the number of configurations explored during the search. We use this metric to indicate the size of the search space explored by each approach, which indicates the search cost. Note that hardware performance counters are used only during training and are not reported as evaluation metrics.

Implementation Details. We implement RLTUNER in Python using the MushroomRL library [36] for the reinforcement learning components. As summarized in Table I, the Q-network is implemented as a Multi-layer Perceptron (MLP) with two hidden layers. The input dimension corresponds to the configuration vector length d, and the output dimension corresponds to the number of possible actions $|\mathcal{A}|$. ReLU is used for the hidden layers, and a linear activation is applied to the output layer. The network is trained using the Adam optimizer with the mean squared error (MSE) loss. We adopt an ϵ-greedy exploration strategy with a linear decay schedule.

We empirically selected the above hyperparameters based on preliminary experiments over a subset of benchmarks and consistently applied them across all evaluations in this section. Training is terminated after a maximum of 30 episodes ($EP_{\max} = 30$). In most cases, the best configuration was discovered within the first 25 episodes, indicating that 30 episodes provide a sufficient budget. While a larger budget may further improve results, 30 episodes strike a practical balance between solution quality and search cost (see Section IV-B).

Experiments were conducted on a workstation hosted on the Chameleon Cloud testbed [37] running Ubuntu 20.04 LTS, equipped with an Intel Xeon Platinum 8380 CPU (2.30 GHz) and 256 GB of RAM.

TABLE II: Configuration count, search cost, speedup, relative error across benchmarks

Suite	Benchmark	Idx	LOC	RLTuner				Precimonious				HiFPTuner			
				Configs	Search Cost	Speedup	Error	Configs	Search Cost	Speedup	Error	Configs	Search Cost	Speedup	Error
GSL	arclength	1	79	137	0:11:02	45.65%	2.91E-05	30	0:01:01	42.88%	7.41E-05	77	0:03:41	0.97%	2.08E-05
	simpsons	2	82	114	0:16:58	4.69%	1.39E-10	107	0:11:07	4.39%	1.39E-10	83	0:07:58	0.03%	3.13E-11
Polybench	correlation	3	153	267	0:09:44	10.19%	7.79E-08	147	0:04:21	1.12%	7.33E-08	83	0:03:23	0.72%	1.22E-08
	covariance	4	131	184	0:05:03	5.87%	1.07E-07	150	0:03:52	1.14%	3.25E-08	57	0:02:24	0.44%	6.75E-08
	2mm	5	151	328	0:19:20	4.96%	1.31E-14	209	0:29:31	1.72%	5.24E-13	61	0:11:27	2.86%	3.91E-08
	3mm	6	156	315	0:18:10	5.61%	2.21E-14	327	0:20:10	6.89%	4.85E-09	67	0:09:04	1.66%	4.85E-09
	bicg	7	133	245	0:06:12	25.27%	1.35E-06	123	0:01:39	0.17%	7.22E-05	107	0:02:15	0.20%	1.25E-06
	gemm	8	128	246	0:16:01	4.40%	6.18E-05	109	0:09:13	1.42%	1.41E-08	57	0:07:30	0.46%	1.41E-08
	gemver	9	144	531	0:14:07	1.63%	2.84E-15	549	0:16:22	1.96%	1.67E-05	141	0:03:22	0.00%	–
	symm	10	124	272	0:24:13	3.69%	1.48E-12	221	0:15:46	3.01%	1.77E-06	67	0:07:31	3.40%	1.77E-06
	syrk	11	129	227	0:02:09	22.62%	1.90E-07	140	0:01:00	5.39%	5.39E-08	31	0:00:16	17.07%	1.41E-08
PARSEC	Blackscholes	12	239	762	0:28:48	12.30%	4.51E-08	922	0:31:40	8.62%	1.32E-06	449	0:40:45	0.00%	–
Rodinia	CFD	13	549	2381	0:18:07	4.24%	1.80E-09	2120	1:56:02	0.80%	7.74E-11	1101	0:52:27	0.00%	–
	Hotspot	14	263	466	0:04:29	5.26%	4.60E-06	235	0:01:38	5.31%	4.60E-06	45	0:00:58	3.78%	4.60E-06
	Kmeans	15	314	274	0:28:38	13.97%	0.00E+00	223	0:34:47	12.48%	0.00E+00	3	0:04:40	0.83%	0.00E+00
	LavaMD	16	253	579	0:02:44	13.91%	0.00E+00	293	0:01:53	3.04%	1.50E-07	191	0:11:50	0.00%	–
NAS	cg	17	829	428	0:50:45	8.36%	3.36E-10	189	0:20:58	0.65%	2.95E-05	141	0:26:27	0.50%	8.64E-06
Mantevo	HPCCG	18	257	663	0:51:08	7.66%	0.00E+00	458	0:10:02	5.33%	0.00E+00	117	0:04:18	0.00%	–
	miniAMR	19	678	895	5:05:53	46.28%	6.97E-15	167	10:10:06	2.71%	6.97E-15	147	3:17:49	8.71%	6.97E-15
LLNL	LULESH	20	2601	2534	22:46:15	17.89%	7.84E-05	3870	34:06:25	5.58%	3.46E-05	1456	8:53:30	10.12%	3.12E-05

B. RQ1: Comparison with Baselines

We compare RLTuner with two baselines, Precimonious and HiFPTuner, in terms of achieved speedup, relative error, and number of explored configurations.

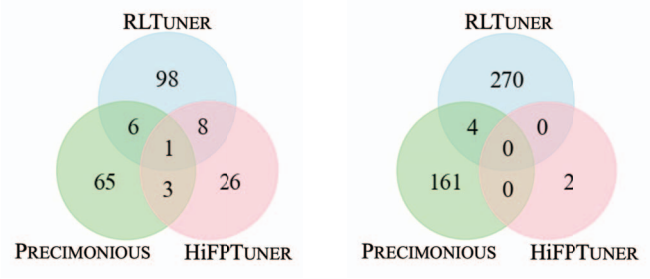
Table II presents a comprehensive comparison between RLTuner and the two baselines across 20 benchmarks under a uniform target error threshold of 10^{-4}, which serves as the default application-level correctness criterion in our evaluation. This threshold is used to ensure a consistent comparison of optimization benefits across benchmarks. For miniAMR, all evaluated configurations already satisfy the 10^{-4} threshold; therefore, we additionally report results under a stricter threshold of 10^{-8} to illustrate behavior under higher accuracy requirements. A sensitivity analysis with respect to different user-specified error thresholds is provided in RQ3.

To ensure methodological rigor, we performed five independent tuning runs for each benchmark, with the exception of the more computationally intensive LULESH and miniAMR, which were repeated three times. We report the peak speedup attained across these iterations. The results demonstrate significant stability: 16 of the 18 benchmarks showed a performance variance below 10%, while LULESH exhibited tighter bounds of 3%. In contrast, miniAMR exhibited higher variability (up to 23%), which can be attributed to its adaptive mesh refinement nature.

Program Speedup. Overall, RLTuner outperforms the baselines in 17 out of 20 benchmarks in terms of program speedup while strictly adhering to the prescribed error thresholds, in both computational kernels and large-scale applications. For instance, in kernels like bicg and syrk, RLTuner achieves speedups of 25.27% and 22.62%, respectively, significantly higher than Precimonious, which achieves only 0.17% and 5.39%. In more complex proxy applications, such as LavaMD, RLTuner delivers a 13.91% speedup, outperforming both baselines and demonstrating its ability to optimize programs with intricate control flows and data dependencies. Similar trends are observed in production-relevant HPC mini-applications, including miniAMR and LULESH, where RLTuner achieves speedups of 46.28% and 17.89%, respectively. These results indicate that the DQN agent is capable of identifying more aggressive yet safe mixed-precision configurations that traditional delta-debugging or dependency-based heuristics overlook.

We note that the speedups observed for Precimonious and HiFPTuner in our environment differ from those reported in their original publications (e.g., for simpsons and arclength). To ensure a rigorous comparison, we utilized the updated re-implementations provided in recent literature [2]. We attribute these performance discrepancies to the evolution of hardware architectures and the maturation of compiler technology. Modern compilers now perform significantly more aggressive optimizations than those available at the time of the original studies, which fundamentally alters the baseline performance landscape.

Numerical Error. We observe that RLTuner does not merely trade precision for performance; rather, it identifies "high-quality" configurations that achieve the best of both performance and accuracy. As highlighted in Table II (noted by orange cells), in 11 out of 20 benchmarks, RLTuner-optimized programs achieve the lowest relative error compared to all baselines, while in 8 benchmarks, RLTuner achieves the highest speedup. For instance, in cg, RLTuner delivers a 8.36% speedup with an error of 3.36E-10, whereas Precimonious achieves only 0.65% speedup while incurring



(a) simpsons (b) Kmeans

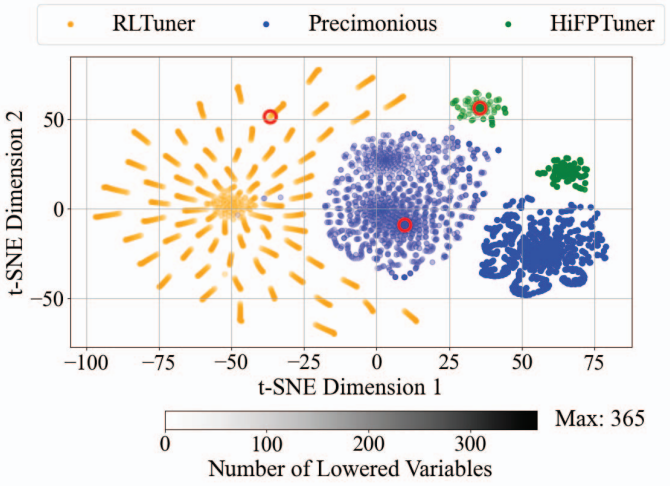
Fig. 3: The number of overlapping configurations between RLTUNER and the baselines

a higher error of 2.95E-05. Compared to PRECIMONIOUS and HIFPTUNER, RLTUNER learns to safely reduce floating-point precision without amplifying error propagation. This indicates that the DQN-based approach can effectively balance performance improvement and numerical stability.

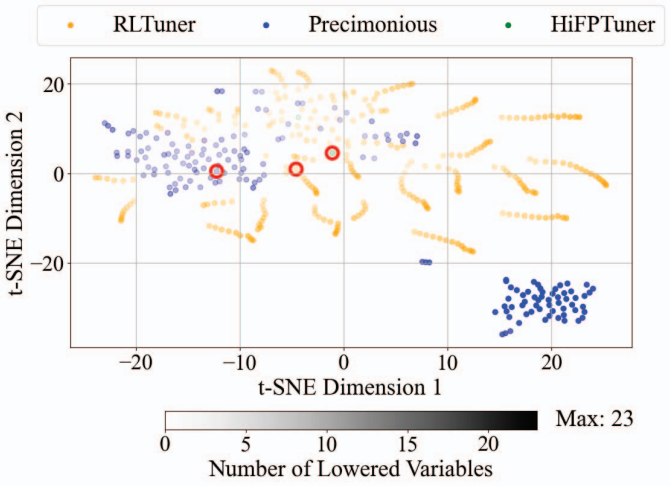
Compared to PRECIMONIOUS and HIFPTUNER, RLTUNER exhibits more stable error behavior across benchmarks. While HIFPTUNER frequently fails to identify precision configurations that yield meaningful speedups, and PRECIMONIOUS occasionally trades off performance for overly conservative precision choices, RLTUNER learns to safely reduce floating-point precision without amplifying error propagation. This indicates that the DQN-based approach can effectively balance performance improvement and numerical stability.

Search Efficiency. RLTUNER demonstrates highly competitive search efficiency compared to traditional methods. As shown in Table II, although RLTUNER occasionally evaluates more configurations to refine its learned policy, the total number of explored configurations remains on the same order of magnitude as PRECIMONIOUS, and in several cases, it even requires fewer evaluations to reach a superior solution. For instance, in Blackscholes, RLTUNER converges using 17% fewer configurations than PRECIMONIOUS while achieving higher performance. In contrast, HIFPTUNER often evaluates fewer configurations due to its dependency-based heuristics, but this efficiency comes at the cost of being trapped in suboptimal solutions, leading to negligible speedup in many benchmarks. While PRECIMONIOUS can converge quickly on small-scale kernels, its struggles with the high-dimensional search spaces of larger applications. Notably, on our largest benchmark, LULESH, RLTUNER identifies a configuration achieving a 17.89% speedup within 22.5 hours, whereas PRECIMONIOUS requires over 34.1 hours, i.e. 33% more time, to reach a lower speedup of 5.58%. This demonstrates the scalability of RLTUNER when tuning larger programs.

Response to RQ1: RLTUNER outperforms baselines in 85% of benchmarks, achieving up to 46.28% speedup while maintaining superior numerical accuracy in 55% of cases, demonstrating its effectiveness in discovering high-quality mixed-precision configurations in complex search spaces.



(a) LULESH



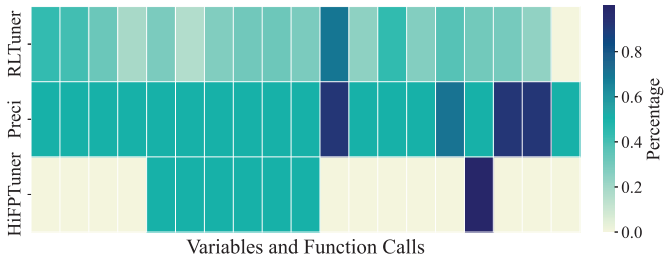
(b) Kmeans

Fig. 4: t-SNE plots of the search spaces of RLTUNER, PRECIMONIOUS, and HIFPTUNER for two benchmarks

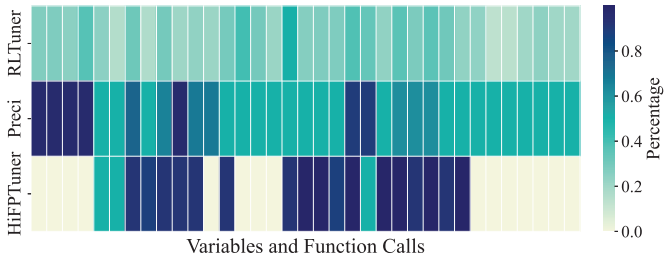
C. RQ2: Space Exploration

This section compares the space exploration of RLTUNER and the baselines in terms of (1) the number of identical configurations explored during the search, (2) the overall search space explored, and (3) the final configurations.

1) Identical Configurations: Overlap between the configurations explored during the search by RLTUNER and the baselines is exceptionally rare. Only one benchmark, simpsons, shows any overlap between RLTUNER, PRECIMONIOUS, and HIFPTUNER. Even then, across the 207 total configurations explored for simpsons (after removing duplicates per tool), only one same configuration was identified by all three methods (Figure 3a). As shown in Figure 3, between RLTUNER and PRECIMONIOUS, overlap is found in only two benchmarks—simpsons (7 configurations) and Kmeans (4 configurations)—with zero overlap elsewhere. RLTUNER and HIFPTUNER share configurations in only simpsons (9 configurations). Furthermore, in covariance, PRECIMONIOUS and HIFPTUNER share only a single configuration, while RLTUNER remains completely isolated. This lack of intersection highlights the



(a) syrk



(b) Hotspot

Fig. 5: Heatmaps of precision-reduced variables and mathematical functions

distinctiveness of the search trajectories produced by each approach.

2) Comparing Search Spaces: To visualize the high-dimensional configuration space, we employ t-SNE to map each binary precision-assignment vector into a two-dimensional plot. Spatial proximity indicates configuration similarity, while the opacity of each dot scales with the number of variables and function calls whose precision is reduced.

Across all benchmarks, a distinct pattern emerges: RLTUNER generates linear trajectories, reflecting the sequential nature of its learning episodes, whereas the baselines produce localized clusters. As seen in the motivating example (Figure 1), this confirms that RLTUNER performs a broader, more continuous exploration of the search space.

This contrast is particularly evident in the LULESH benchmark (Figure 4a). RLTUNER produces visible "episode paths" that span the space, while PRECIMONIOUS remains confined to discrete, binary-style clusters and HIFPTUNER forms small, dense communities. Even in lower-density spaces like Kmeans (Figure 4b), where all methods appear more dispersed, RLTUNER maintains superior coverage while successfully targeting the most promising configurations. In RQ1, we already showed that RLTUNER achieves better speedup than PRECIMONIOUS and HIFPTUNER in most benchmarks.

3) Targeted Variables and Functions: To analyze variable-level exploration, we utilize the heatmaps in Figure 5, which visualize the frequency with which each variable is assigned lower precision across all explored configurations. In these plots, color intensity correlates with the reduction ratio: darker cells represent variables frequently targeted for tuning, while lighter cells indicate those rarely modified. This visualization confirms that RLTUNER and the baselines target distinct, non-overlapping subsets of variables and function calls. These patterns remain consistent across benchmarks regardless of

TABLE III: Error thresholds vs. performance & precision

	Benchmark	RLTUNER		PRECIMONIOUS		HIFPTUNER	
		Speedup	Error	Speedup	Error	Speedup	Error
10^{-4}	syrk	22.62%	1.90E-07	5.39%	5.39E-08	17.07%	1.41E-08
	Hotspot	5.26%	4.60E-06	5.31%	4.60E-06	3.78%	4.60E-06
	LavaMD	13.91%	0.00E+00	3.04%	1.50E-07	0.00%	–
	cg	8.36%	3.36E-10	0.65%	2.95E-05	0.50%	8.64E-06
	LULESH	17.89%	7.84E-05	5.58%	3.46E-05	10.12%	3.12E-05
10^{-6}	syrk	11.33%	5.39E-08	3.35%	1.41E-08	0.27%	1.41E-08
	Hotspot	2.62%	6.53E-12	1.75%	2.51E-11	0.41%	1.22E-14
	LavaMD	8.36%	0.00E+00	0.17%	0.00E+00	0.00%	–
	cg	0.59%	3.26E-09	0.57%	3.26E-09	0.58%	1.23E-08
	LULESH	30.49%	5.82E-11	6.72%	3.00E-11	17.79%	6.30E-07
10^{-8}	syrk	24.86%	6.08E-15	7.82%	6.08E-15	4.98%	6.08E-15
	Hotspot	3.45%	1.33E-12	1.95%	2.51E-11	0.28%	2.37E-11
	LavaMD	8.40%	0.00E+00	0.00%	–	8.08%	0.00E+00
	cg	0.55%	3.36E-10	0.52%	3.26E-09	0.53%	3.26E-09
	LULESH	56.39%	5.09E-11	20.57%	1.75E-09	17.96%	2.55E-13

search space complexity. We highlight syrk and Hotspot as representative examples, spanning theoretical search spaces of sizes 2^{19} and 2^{35}.

Figure 5a shows that each approach lowers different sets of variables in syrk. RLTUNER mainly targets `sqrt`, `A`, and `alpha`, whose precision is reduced in 69%, 43%, and 42% of the configurations, respectively. In contrast, PRECIMONIOUS focuses on `sqrt`, `sin` in 91% of the configurations, and `beta` in 70% of the configurations. HIFPTUNER concentrates reductions more locally, with `checksum` and `C` lowered in 100% and 50% of the configurations, respectively.

In Hotspot (Figure 5b), PRECIMONIOUS reduces the precision of `amb_temp_1`, `chip_height`, and `t_chip` in nearly 97% of the configurations, while RLTUNER distributes reductions more broadly, with `grid_height`, `result`, and `Ry` (51%, 40%, and 38%). HIFPTUNER lowers `Ry`, `Ry_1`, and `Cap_1`, though very intensively. Overall, RLTUNER selects variables and function calls with higher potential to improve performance and using error as feedback, unlike the static or hierarchically structured strategies of PRECIMONIOUS and HIFPTUNER.

Response to RQ2: RLTUNER and the baselines explore different parts of the search space, barely overlapping in their explored configurations. RLTUNER explores a larger search space than PRECIMONIOUS and HIFPTUNER, and it has the potential to find configurations with higher speedup and smaller relative error. The approaches also differ on the entities whose precision are reduced across most configurations.

D. RQ3: Sensitivity to Error Thresholds

To assess the sensitivity of RLTUNER to varying accuracy requirements, we evaluate its performance under three increasingly stringent error thresholds commonly used in the precision tuning literature: 10^{-4}, 10^{-6}, and 10^{-8}. We select five representative benchmarks including one numerical kernel, two proxy applications, and two production-relevant

TABLE IV: Generalizability analysis across different datasets and problem sizes. Highlighted cells denote the representative input used during tuning. ✓indicates the optimal configuration satisfies the error threshold on unseen inputs.

Suite	Benchmark	Input	Pass	Error	Speedup
Polybench	bicg	STANDARD_DATASET	✓	4.87E-09	17.08%
		LARGE_DATASET	✓	5.15E-09	17.51%
		EXTRALARGE_DATASET	✓	1.35E-06	18.63%
	symm	SMALL_DATASET	✓	6.56E-12	27.08%
		STANDARD_DATASET	✓	1.48E-12	18.34%
		LARGE_DATASET	✓	6.31E-13	9.40%
	syrk	SMALL_DATASET	✓	1.21E-05	42.28%
		STANDARD_DATASET	✓	1.90E-07	49.14%
		LARGE_DATASET	✓	4.92E-08	38.00%
Rodinia	CFD	fvcorr.domn.097K	✓	1.80E-09	4.11%
		fvcorr.domn.193K	✓	2.01E-14	2.63%
	Hotspot	1024x1024, grid rows x cols	✓	4.60E-06	46.29%
		2048x2048, grid rows x cols	✓	3.08E-10	22.17%
NAS	cg	CLASS S	✓	2.95E-07	0.52%
		CLASS W	✓	4.46E-07	2.30%
		CLASS A	✓	3.36E-10	4.43%
		CLASS B	✓	4.62E-07	7.06%
		CLASS C	✓	1.05E-05	7.86%
LLNL	LULESH	mesh_size = 30	✓	7.84E-05	19.30%
		mesh_size = 50	✓	1.48E-05	7.48%
		mesh_size = 70	✓	3.63E-05	5.41%

HPC mini-applications. This analysis complements RQ1 by demonstrating how different tuning approaches adapt when users specify tighter correctness constraints.

As summarized in Table III, RLTUNER maintains a consistent performance lead across all thresholds, achieving the highest speedup in 14 out of 15 scenarios. While speedups naturally decrease as constraints become stricter, RLTUNER maintains a higher residual performance compared to baselines. For instance, in Hotspot, although speedup drops sharply across all tools, RLTUNER sustains a 3.45% gain at 10^{-8}, whereas HIFPTUNER nearly collapses to 0.28%. In another example, syrk with a 10^{-8} threshold, RLTUNER identifies a 24.86% speedup, whereas PRECIMONIOUS and HIFPTUNER achieve 7.82% and 4.98% speedup, respectively. Notably, RLTUNER achieves a higher speedup when the error threshold is 10^{-8} (24.86%) than with the more relaxed 10^{-6} error threshold (11.33%). This non-monotonic improvement demonstrates that RLTUNER's global exploration can uncover high-performing, non-obvious configurations that the greedy heuristics of baselines consistently overlook.

Response to RQ3: RLTUNER exhibits high robustness to varying accuracy requirements, consistently outperforming baselines across all the considered error thresholds. It remains effective under strict constraints where traditional heuristics often fail, confirming its reliability for diverse scientific computing needs.

E. RQ4: Generalizability Analysis

A critical requirement for the practical deployment of automated tuning tools is the generalizability of the identified mixed-precision configurations across varying inputs. In typical HPC workflows, applications are repeatedly executed in production pipelines. Therefore, RLTUNER follows the established methodology of optimizing on a representative workload to extract a highly performant configuration that can then be deployed for long-term production use.

To empirically evaluate the generalizability of RLTUNER, we conducted an analysis using 7 representative benchmarks across different suites (Polybench, Rodinia, NAS, and LLNL). We applied the optimal configurations—identified using the default tuning inputs—to a series of unseen datasets and varying problem sizes. The results are summarized in Table IV, where highlighted cells indicates that the input was used during the tuning process.

The results reveal three key observations. First, in 100% of the cases tested on unseen data, the configurations identified by RLTUNER remained strictly within the pre-defined error threshold. This demonstrates that RLTUNER effectively identifies variables whose precision can be safely reduced based on the underlying algorithmic structure, rather than merely overfitting to the numerical values of a specific input. Second, RLTUNER delivers consistent speedups across all unseen inputs, though the magnitude of the performance gain varies with problem size. For example, in NAS `cg`, the speedup improves from 4.43% (Class A) to 7.86% (Class C) as the problem size scales up. Conversely, in benchmarks like Polybench `symm`, the relative speedup decreases on larger datasets. We attribute these variations to shifting architectural bottlenecks: as problem sizes scale, applications may transition between being compute-bound and memory-bound, altering the relative performance benefits of reduced precision memory accesses versus arithmetic operations. Third, the generalizability holds even for complex, large-scale mini-apps like LULESH. When increasing the mesh size from 30 to 70—representing a nearly 13-fold increase in the total computational workload—the configuration identified by RLTUNER consistently yields performance gains (7.48% for a 50^3 mesh and 5.41% for a 70^3 mesh) without any accuracy violations.

While dynamic tuning tools cannot theoretically guarantee numerical accuracy for all conceivable inputs, our results strongly suggest that by leveraging reinforcement learning to explore a wider and more diverse configuration space during training, RLTUNER discovers robust configurations that exhibit strong generalizability across input variations and scaling workloads.

Response to RQ4: RLTUNER demonstrates strong generalizability across diverse inputs and problem sizes. Configurations tuned on representative workloads achieve 100% adherence to error thresholds on unseen data. Furthermore, they deliver sustained performance speedups—scaling efficiently even with up to 13× workload increases—without overfitting to the initial tuning inputs.

V. Threats to Validity

a) External Validity: RLTuner utilizes DQN, selected for its proven efficacy in goal-directed learning as a foundational model-free algorithm [38]; while results may not generalize to other RL architectures, we mitigated this by evaluating on a diverse suite of widely recognized benchmarks. Our implementation targets C/C++ with LLVM, but only the source-level rewriter is language-specific—the RL-based search, reward computation, and convergence criteria are language- and compiler-agnostic, making adoption to other compilers or languages possible. To address stochasticity, we repeated each experiment $3\times$ for LULESH and miniAMR and $5\times$ for the remaining benchmarks; in 16 of 18 cases, speedup remained within a 10% range, with LULESH showing only 3% variation, while miniAMR exhibited 23% variation due to its sensitive mesh refinement nature.

While RLTuner is designed to be hardware-agnostic and independent of specific floating-point formats, our current evaluation focuses on FP64-to-FP32 transformations on CPU platforms. This scope may not fully capture performance and numerical behaviors inherent to GPUs and other AI accelerators, which frequently utilize formats like FP16, BF16, TF32, or FP8 and exhibit different execution characteristics. Nevertheless, adapting RLTuner to these targets is architecturally feasible. Since the RL-based search and convergence criteria are platform-independent, only the source-level rewriter and the reward signal require extension. Although our reward function can default to execution time ($\beta = 0$) when detailed performance metrics are unavailable, capturing accelerator-specific factors such as tensor-core utilization and memory-bandwidth savings may require additional reward components to faithfully represent optimization trade-offs. Consequently, further validation across diverse architectures is required to confirm the generality of our approach.

Another promising direction involves incorporating energy efficiency, which is increasingly critical given the rising power demands of large-scale HPC and AI workloads. Since RL-Tuner utilizes a multi-objective reward function, integrating an energy term (e.g., derived from RAPL counters on CPUs or NVML on GPUs) is a natural extension. This would enable users to identify optimal configurations that balance performance, accuracy, and energy consumption.

b) Internal Validity: We address several threats to internal validity in our experimental design. First, to mitigate runtime variance, we execute each configuration multiple times—$5\times$ for LULESH and $10\times$ for all other benchmarks—and report the average speedup. Second, to ensure performance results are not skewed by varying compiler toolchains, we utilize the default compiler settings provided by each benchmark suite across all experiments. Finally, as with all dynamic precision tuners, effectiveness is contingent upon the input sets used during tuning. Following established practices [2], [5], [9], we assume the provided inputs are sufficiently representative, and further evaluate generalizability on unseen inputs (see RQ4).

VI. Related Work

Dynamic analysis is widely utilized for tuning floating-point precision. Search-based tools such as Precimonious [9] and HiFPTuner [5] employ delta-debugging and hierarchical search, respectively, prioritizing aggressive performance gains over sound error guarantees [30]. Blame Analysis [13] prunes the search space via shadow execution, while ADAPT/FloatSmith [4], [33] analyze workload sensitivity at the source level. Other frameworks target alternative representations [10], [34], [39] or specific application domains, such as climate modeling [7]. On the conservative side, Daisy [11], Rosa [40], and Salsa [41] provide sound accuracy guarantees through static analysis but often struggle with complex control flows. More recently, FPLearner [2] uses graph neural networks to predict performance and accuracy. Unlike these tools, RLTuner treats accuracy as a first-class objective by incorporating relative error directly into the reward function.

A significant body of work targets GPU-based precision tuning. AMPT-GA [28] leverages genetic algorithms for GPU kernel variables, GPU-FPTuner [32] introduces automated planning for mixed-precision GPU applications, and other efforts address floating-point registers [42] and GPU kernels [43]. In contrast, our approach is platform-agnostic and applicable to a broader range of environments.

RL has emerged as a powerful paradigm for complex optimization, including database tuning [21], quadratic programming [18], and network optimization [19], [20]. DRL has proven especially effective for sequential decision-making in compiler optimization [44], [45], cloud resource provisioning [46], and multi-core energy management [47]. While these works demonstrate the versatility of DRL in systems optimization, ours is the first to leverage it for mixed-precision floating-point tuning.

VII. Conclusion

We presented RLTuner, a Deep Reinforcement Learning-based approach for precision tuning that leverages DQN to balance speedup and accuracy. By formulating the tuning task as a Markov Decision Process, RLTuner enables learned, reward-driven exploration of the search space. Across 20 benchmarks, RLTuner outperforms two state-of-the-art baselines, Precimonious and HiFPTuner, in 17 benchmarks, achieving speedups of up to 46.28% and yielding the lowest relative error in 11 benchmarks. RLTuner is publicly available at https://github.com/ucd-plse/RLTuner.

VIII. Acknowledgments

This work was supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research, under award DE-SC0020286, and the National Science Foundation under awards CNS-2346396 and CCF-2119348. Results presented in this paper were obtained using the Chameleon testbed supported by the National Science Foundation.

References

[1] D. B. Khalifa and M. Martel, "Everything you need to know about reduced mixed precision computation in numerical programs," 2023.

[2] Y. Wang and C. Rubio-González, "Predicting performance and accuracy of mixed-precision programs for precision tuning," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 2024, pp. 15:1–15:13. [Online]. Available: https://doi.org/10.1145/3597503.3623338

[3] M. O. Lam, J. K. Hollingsworth, B. R. de Supinski, and M. P. LeGendre, "Automatically adapting programs for mixed-precision floating-point computation," in *International Conference on Supercomputing, ICS'13, Eugene, OR, USA - June 10 - 14, 2013*, A. D. Malony, M. Nemirovsky, and S. P. Midkiff, Eds. ACM, 2013, pp. 369–378. [Online]. Available: https://doi.org/10.1145/2464996.2465018

[4] M. O. Lam, T. Vanderbruggen, H. Menon, and M. Schordan, "Tool integration for source-level mixed precision," in *2019 IEEE/ACM 3rd International Workshop on Software Correctness for HPC Applications (Correctness), Denver, CO, USA, November 18, 2019*, I. Laguna and C. Rubio-González, Eds. IEEE, 2019, pp. 27–35. [Online]. Available: https://doi.org/10.1109/Correctness49594.2019.00009

[5] H. Guo and C. Rubio-González, "Exploiting community structure for floating-point precision tuning," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, F. Tip and E. Bodden, Eds. ACM, 2018, pp. 333–343. [Online]. Available: https://doi.org/10.1145/3213846.3213862

[6] S. Cherubin and G. Agosta, "Tools for reduced precision computation: A survey," *ACM Comput. Surv.*, vol. 53, no. 2, pp. 33:1–33:35, 2021. [Online]. Available: https://doi.org/10.1145/3381039

[7] J. Vanover, A. Altuntas, and C. Rubio-González, "Toward automated precision tuning of weather and climate models: A case study," in *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis, Atlanta, GA, USA, November 17-22, 2024*. IEEE, 2024, pp. 148–159. [Online]. Available: https://doi.org/10.1109/SCW63240.2024.00026

[8] W. Chiang, M. Baranowski, I. Briggs, A. Solovyev, G. Gopalakrishnan, and Z. Rakamaric, "Rigorous floating-point mixed-precision tuning," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, G. Castagna and A. D. Gordon, Eds. ACM, 2017, pp. 300–315. [Online]. Available: https://doi.org/10.1145/3009837.3009846

[9] C. Rubio-González, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough, "Precimonious: tuning assistant for floating-point precision," in *International Conference for High Performance Computing, Networking, Storage and Analysis, SC'13, Denver, CO, USA - November 17 - 21, 2013*, W. Gropp and S. Matsuoka, Eds. ACM, 2013, pp. 27:1–27:12. [Online]. Available: https://doi.org/10.1145/2503210.2503296

[10] S. Graillat, F. Jézéquel, R. Picot, F. Févotte, and B. Lathuilière, "Auto-tuning for floating-point precision with discrete stochastic arithmetic," *J. Comput. Sci.*, vol. 36, 2019. [Online]. Available: https://doi.org/10.1016/j.jocs.2019.07.004

[11] E. Darulova, E. Horn, and S. Sharma, "Sound mixed-precision optimization with rewriting," in *Proceedings of the 9th ACM/IEEE International Conference on Cyber-Physical Systems, ICCPS 2018, Porto, Portugal, April 11-13, 2018*, C. Gill, B. Sinopoli, X. Liu, and P. Tabuada, Eds. IEEE Computer Society / ACM, 2018, pp. 208–219. [Online]. Available: https://doi.org/10.1109/ICCPS.2018.00028

[12] H. Brunie, C. Iancu, K. Z. Ibrahim, P. Brisk, and B. Cook, "Tuning floating-point precision using dynamic program information and temporal locality," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020*, C. Cuicchi, I. Qualters, and W. T. Kramer, Eds. IEEE/ACM, 2020, p. 50. [Online]. Available: https://doi.org/10.1109/SC41405.2020.00054

[13] C. Rubio-González, C. Nguyen, B. Mehne, K. Sen, J. Demmel, W. Kahan, C. Iancu, W. Lavrijsen, D. H. Bailey, and D. Hough, "Floating-point precision tuning using blame analysis," in *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, L. K. Dillon, W. Visser, and L. A. Williams, Eds. ACM, 2016, pp. 1074–1085. [Online]. Available: https://doi.org/10.1145/2884781.2884850

[14] K. Parasyris, I. Laguna, H. Menon, M. Schordan, D. Osei-Kuffuor, G. Georgakoudis, M. O. Lam, and T. Vanderbruggen, "Hpc-mixpbench: An HPC benchmark suite for mixed-precision analysis," in *IEEE International Symposium on Workload Characterization, IISWC 2020, Beijing, China, October 27-30, 2020*. IEEE, 2020, pp. 25–36. [Online]. Available: https://doi.org/10.1109/IISWC50251.2020.00012

[15] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization, IISWC 2009, October 4-6, 2009, Austin, TX, USA*. IEEE Computer Society, 2009, pp. 44–54. [Online]. Available: https://doi.org/10.1109/IISWC.2009.5306797

[16] R. S. Sutton and A. G. Barto, "Reinforcement learning: An introduction," *IEEE Trans. Neural Networks*, vol. 9, no. 5, pp. 1054–1054, 1998. [Online]. Available: https://doi.org/10.1109/TNN.1998.712192

[17] M. van Otterlo and M. A. Wiering, "Reinforcement learning and markov decision processes," in *Reinforcement Learning*, ser. Adaptation, Learning, and Optimization, M. A. Wiering and M. van Otterlo, Eds. Springer, 2012, vol. 12, pp. 3–42. [Online]. Available: https://doi.org/10.1007/978-3-642-27645-3_1

[18] J. Ichnowski, P. Jain, B. Stellato, G. Banjac, M. Luo, F. Borrelli, J. E. Gonzalez, I. Stoica, and K. Goldberg, "Accelerating quadratic optimization with reinforcement learning," pp. 21 043–21 055, 2021. [Online]. Available: https://proceedings.neurips.cc/paper/2021/hash/afdec7005cc9f14302cd0474fd0f3c96-Abstract.html

[19] N. Jiang, Y. Deng, A. Nallanathan, and J. A. Chambers, "Reinforcement learning for real-time optimization in nb-iot networks," *IEEE J. Sel. Areas Commun.*, vol. 37, no. 6, pp. 1424–1440, 2019. [Online]. Available: https://doi.org/10.1109/JSAC.2019.2904366

[20] M. N. ul Islam and A. Mitschele-Thiel, "Cooperative fuzzy q-learning for self-organized coverage and capacity optimization," in *23rd IEEE International Symposium on Personal, Indoor and Mobile Radio Communications, PIMRC 2012, Sydney, Australia, September 9-12, 2012*. IEEE, 2012, pp. 1406–1411. [Online]. Available: https://doi.org/10.1109/PIMRC.2012.6362568

[21] J. Wang, I. Trummer, and D. Basu, "UDO: universal database optimization using reinforcement learning," *Proc. VLDB Endow.*, vol. 14, no. 13, pp. 3402–3414, 2021. [Online]. Available: http://www.vldb.org/pvldb/vol14/p3402-wang.pdf

[22] F. U. Haq, D. Shin, and L. C. Briand, "Many-objective reinforcement learning for online testing of dnn-enabled systems," in *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2023, pp. 1814–1826. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00155

[23] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, "Auto-tuning a high-level language targeted to gpu codes," in *2012 innovative parallel computing (InPar)*. Ieee, 2012, pp. 1–10.

[24] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: characterization and architectural implications," in *17th International Conference on Parallel Architectures and Compilation Techniques, PACT 2008, Toronto, Ontario, Canada, October 25-29, 2008*, A. Moshovos, D. Tarditi, and K. Olukotun, Eds. ACM, 2008, pp. 72–81. [Online]. Available: https://doi.org/10.1145/1454115.1454128

[25] D. Bailey, T. Harris, W. Saphir, R. Van Der Wijngaart, A. Woo, and M. Yarrow, "The nas parallel benchmarks 2.0," Citeseer, Tech. Rep., 1995.

[26] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, "Improving Performance via Mini-applications," Sandia National Laboratories, Tech. Rep. SAND2009-5574, 2009.

[27] I. Karlin, J. Keasler, and J. R. Neely, "Lulesh 2.0 updates and changes," Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), Tech. Rep., 2013.

[28] P. V. Kotipalli, R. Singh, P. Wood, I. Laguna, and S. Bagchi, "AMPT-GA: automatic mixed precision floating point tuning for GPU applications," in *Proceedings of the ACM International Conference on Supercomputing, ICS 2019, Phoenix, AZ, USA, June 26-28, 2019*, R. Eigenmann, C. Ding, and S. A. McKee, Eds. ACM, 2019, pp. 160–170. [Online]. Available: https://doi.org/10.1145/3330345.3330360

[29] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Trans. Software Eng.*, vol. 28, no. 2, pp. 183–200, 2002. [Online]. Available: https://doi.org/10.1109/32.988498

[30] A. Isychev and D. Lohar, "Cost of soundness in mixed-precision tuning," *Proc. ACM Program. Lang.*, vol. 9, no. OOPSLA2, pp. 2426–2453, 2025. [Online]. Available: https://doi.org/10.1145/3763137

[31] S. Cherubin, D. Cattaneo, M. Chiari, A. D. Bello, and G. Agosta, "TAFFO: tuning assistant for floating to fixed point optimization," *IEEE Embed. Syst. Lett.*, vol. 12, no. 1, pp. 5–8, 2020. [Online]. Available: https://doi.org/10.1109/LES.2019.2913774

[32] R. Gu and M. Becchi, "Gpu-fptuner: Mixed-precision auto-tuning for floating-point applications on GPU," in *27th IEEE International Conference on High Performance Computing, Data, and Analytics, HiPC 2020, Pune, India, December 16-19, 2020*. IEEE, 2020, pp. 294–304. [Online]. Available: https://doi.org/10.1109/HiPC50609.2020.00043

[33] H. Menon, M. O. Lam, D. Osei-Kuffuor, M. Schordan, S. Lloyd, K. M. Mohror, and J. Hittinger, "ADAPT: algorithmic differentiation applied to floating-point precision tuning," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC 2018, Dallas, TX, USA, November 11-16, 2018*. IEEE / ACM, 2018, pp. 48:1–48:13. [Online]. Available: http://dl.acm.org/citation.cfm?id=3291720

[34] S. Bessaï, D. Ben Khalifa, H. Benmaghnia, and M. Martel, "Fixed-point code synthesis based on constraint generation," in *Design and Architecture for Signal and Image Processing - 15th International Workshop, DASIP 2022, Budapest, Hungary, June 20-22, 2022, Proceedings*, ser. Lecture Notes in Computer Science, K. Desnos and S. A. Pertuz, Eds., vol. 13425. Springer, 2022, pp. 108–120. [Online]. Available: https://doi.org/10.1007/978-3-031-12748-9_9

[35] N. Ho, E. Manogaran, W. Wong, and A. Anoosheh, "Efficient floating point precision tuning for approximate computing," in *22nd Asia and South Pacific Design Automation Conference, ASP-DAC 2017, Chiba, Japan, January 16-19, 2017*. IEEE, 2017, pp. 63–68. [Online]. Available: https://doi.org/10.1109/ASPDAC.2017.7858297

[36] C. D'Eramo, D. Tateo, A. Bonarini, M. Restelli, and J. Peters, "Mushroomrl: Simplifying reinforcement learning research," *Journal of Machine Learning Research*, vol. 22, no. 131, pp. 1–5, 2021. [Online]. Available: http://jmlr.org/papers/v22/18-056.html

[37] K. Keahey, J. Anderson, Z. Zhen, P. Riteau, P. Ruth, D. Stanzione, M. Cevik, J. Colleran, H. S. Gunawi, C. Hammock, J. Mambretti, A. Barnes, F. Halbach, A. Rocha, and J. Stubbs, "Lessons learned from the chameleon testbed," in *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*. USENIX Association, July 2020.

[38] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. A. Riedmiller, A. 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," *Nat.*, vol. 518, no. 7540, pp. 529–533, 2015. [Online]. Available: https://doi.org/10.1038/nature14236

[39] S. Cherubin, D. Cattaneo, M. Chiari, and G. Agosta, "Dynamic precision autotuning with TAFFO," *ACM Trans. Archit. Code Optim.*, vol. 17, no. 2, pp. 10:1–10:26, 2020. [Online]. Available: https://doi.org/10.1145/3388785

[40] E. Darulova and V. Kuncak, "Towards a compiler for reals," *ACM Trans. Program. Lang. Syst.*, vol. 39, no. 2, pp. 8:1–8:28, 2017. [Online]. Available: https://doi.org/10.1145/3014426

[41] N. Damouche and M. Martel, "Salsa: An automatic tool to improve the numerical accuracy of programs," vol. 5, pp. 63–76, 2017. [Online]. Available: https://doi.org/10.29007/j2fd

[42] A. Angerd, E. Sintorn, and P. Stenström, "A framework for automated and controlled floating-point accuracy reduction in graphics applications on gpus," *ACM Trans. Archit. Code Optim.*, vol. 14, no. 4, pp. 46:1–46:25, 2017. [Online]. Available: https://doi.org/10.1145/3151032

[43] R. Nobre, L. Reis, J. Bispo, T. Carvalho, J. M. P. Cardoso, S. Cherubin, and G. Agosta, "Aspect-driven mixed-precision tuning targeting gpus," in *Proceedings of the 9th Workshop on Parallel Programming and RunTime Management Techniques for Manycore Architectures and 7th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms, PARMA-DITAM@HiPEAC 2018, Manchester, United Kingdom, January 23-23, 2018*. ACM, 2018, pp. 26–31. [Online]. Available: https://doi.org/10.1145/3183767.3183776

[44] R. Mammadli, A. Jannesari, and F. Wolf, "Static neural compiler optimization via deep reinforcement learning," vol. abs/2008.08951, 2020. [Online]. Available: https://arxiv.org/abs/2008.08951

[45] D. R. Lamouri, I. N. Aouadj, S. Kourta, and R. Baghdadi, "Pearl: Automatic code optimization using deep reinforcement learning," in *Proceedings of the 39th ACM International Conference on Supercomputing, ICS 2025, Salt Lake City, UT, USA, June 8-11, 2025*. ACM, 2025, pp. 959–974. [Online]. Available: https://doi.org/10.1145/3721145.3725766

[46] M. Cheng, J. Li, and S. Nazarian, "Drl-cloud: Deep reinforcement learning-based resource provisioning and task scheduling for cloud service providers," in *23rd Asia and South Pacific Design Automation Conference, ASP-DAC 2018, Jeju, Korea (South), January 22-25, 2018*, Y. Shin, Ed. IEEE, 2018, pp. 129–134. [Online]. Available: https://doi.org/10.1109/ASPDAC.2018.8297294

[47] Z. Tian, L. Chen, X. Li, J. Feng, and J. Xu, "Multi-core power management through deep reinforcement learning," in *IEEE International Symposium on Circuits and Systems, ISCAS 2021, Daegu, South Korea, May 22-28, 2021*. IEEE, 2021, pp. 1–5. [Online]. Available: https://doi.org/10.1109/ISCAS51556.2021.9401447