Reinforcement Learning as a Strategy for Kernel Tuning

Alexander Koops (s4061934)

Supervisor:
Ben van Werkhoven & Thomas Moerland

Bachelor Thesis – Informatica

Leiden Institute of Advanced Computer Science (LIACS)
liacs.leidenuniv.nl

July 13, 2026

Abstract

In GPU programming, kernel performance is dependent on both GPU architecture and the kernel hyperparameters. Hyperparameter configurations that work well on a specific kernel and GPU combination may perform poor on another GPU and kernel combination. This is why auto-tuners exist, implementing a variety of optimization algorithms to tune any combination. Reinforcement learning (RL) approaches exist for this tuning problem, but are all LLM-based, focusing on generating the entire kernel code. No RL approach exists for tuning existing kernel code, to the best of our knowledge. This work develops two training environments, two different PPO-based agents and integrates these agents with Kernel Tuner to evaluate their performance against existing strategies. The key findings show a strong early performance of RL-based agents, compared to existing optimization strategies, but that the agents can not compete with the consistent improvement of these existing strategies. The strong early performance of the RL agents demonstrates the potential of this approach, establishing a promising foundation for future work on RL-based optimization strategies for GPU kernel tuning.

Contents

1 Introduction

GPUs have become a central component in scientific research, machine learning, and HPC. In GPU programming, the performance of the kernels depends on the tunable parameters and the underlying architecture. These two aspects make it extremely difficult to find the best performing combination of parameters for a given GPU architecture, since all possible combinations can result in a very large search space. Finding a near-optimal combination of parameters is important to preserve resources such as time, energy, and costs.

Since the search space can become very large, it is infeasible for a programmer to tune a kernel by hand, which is why Auto-Tuners such as Kernel Tuner (KT) [34] and OpenTuner [3] exist. These auto-tuners implement optimization algorithms to find a near-optimum configuration. This work focuses on KT, which has different optimization algorithms implemented as tuning strategies. These strategies start without any prior knowledge of the search space and execute the optimization. This creates the question: what if prior knowledge can be used to improve auto-tuning? This is a gap that reinforcement learning (RL) can potentially leverage, by using the experience of prior runs. While RL has been applied to GPU kernel optimization, these existing approaches are based on LLMs and generate new kernels, rather than tuning the existing ones. This work addresses this gap by implementing RL as a direct optimization strategy within KT.

This work enables us to answer the following:

Research Question: How do PPO-based agents with different action spaces perform compared to existing optimization strategies in Kernel Tuner for auto-tuning GPU kernels?

To answer this research question, this work takes the following steps. First, a Gymnasium [5] based RL environment is developed with two action space variants: a Hamming-1 action space and a per-parameter value selection action space. Second, two PPO-based [26] agents are developed, one using a Hamming-1 action space and one using a per-parameter action space. For each agent type, a separate model is trained per kernel, resulting in kernel specific agents that learn to navigate that kernel's search space. Each agent is trained on pre-computed cache files from the AutoTuningAssociation benchmark hub, covering four GPU kernels over multiple GPU architectures. This results in a total of 8 trained agents. Third, the trained agents are integrated with Kernel Tuner [34] as an optimization strategy. Lastly, the agents are evaluated on held-out GPU architectures not seen during training, comparing performance over wall-clock time and function evaluations against random search, Basin Hopping, Genetic Algorithm, Differential Evolution, and Bayesian Optimization, following the comparison methodology from Willemsen et al. [36].

1.1 Thesis overview

This bachelor thesis was conducted at the Leiden Institute of Advanced Computer Science (LIACS), under the supervision of Ben van Werkhoven and Thomas Moerland. The thesis is structured as follows. Section 2 provides necessary background. Section 3 discusses related work. Section 4 discusses the environment design, agent training procedure and the experiment methodology. Section 5 describes the experiments and their outcomes. Section 6 concludes the research question and discusses possible future work.

2 Background

This section contains the necessary background information needed to understand the remainder of this thesis. It covers GPU programming, kernel tuning, the kernels used in this research, the optimization strategies used for comparisons, and the reinforcement learning concepts used for the agent development.

2.1 GPU Programming and Architecture

The Graphic Processing Unit (GPU) is a processing chip, specialized for highly parallel computations. Therefore, they have a different architecture compared to the classic Central Processing Unit (CPU), which are optimized for sequential computations.

GPU architectures consist of different components, working together to create the parallel processing power. The core pieces of the GPU, which play a similar role in the GPU as available cores do in a CPU, are the Compute Units (CUs), named by NVIDIA as: Streaming Multiprocessors (SMs) [17]. These SMs consist of a number of functional units, which are the components responsible for executing computations. The SMs also have a local register file and unified data cache, providing the physical resources for the L1 cache and shared memory. Across GPU architectures, memory sizes and functional unit counts vary, resulting in a wide range of SM configurations.

To run code on these GPUs, programmers write kernels. These kernels are code pieces, specifically written for GPUs in languages like CUDA [17] or OpenCL [27]. These kernels have different tunable parameters, like block size or loop unrolling factors, whose influence on performance is hardware dependent.

2.2 Kernel Tuning

Kernel performance depends on both tunable parameters and the underlying GPU architecture, creating the need for programmers to search for an optimal solution. The problem with tuning these parameters is that the amount of valid combinations, also called the search space, can reach millions, as each additional parameter multiplies the total number of combinations by its number of options. This makes the search space scale as a product of the total parameter options.

The search space that the kernel parameters create can be pictured as a landscape, where every configuration of parameter values is one dot, with the height of the dot in the landscape being the execution time, where a lower dot represents a faster execution time. A simple example of what this might look like is given in Figure 1.

To optimally tune the kernel, the lowest point in this landscape has to be found, since the execution time is at its minimum there.

2.3 Kernel Tuner

To make kernel tuning accessible, auto-tuners have been developed. Auto-tuners, such as OpenTuner [3] and Kernel Tuner [34], are programs designed to find the optimal parameter combination within a set amount of evaluations. In this thesis, Kernel Tuner is the auto-tuner of choice.

Kernel Tuner is an open-source, easy-to-use auto-tuner for CUDA [17], OpenCL [27] and C kernels. By providing a Python API, Kernel Tuner enables programmers to write small python

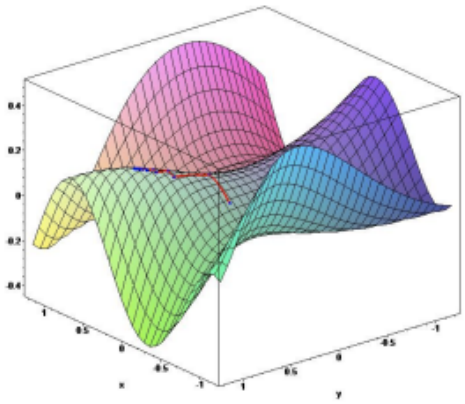


Figure 1: Example landscape [28].

scripts to tune their kernel, in which they provide the location of the to-be-tuned kernel code, how to call this kernel code including input data and the tunable parameters. After which, Kernel Tuner is able to handle the compiling, benchmarking and searching for the best performing kernel configuration itself.

Kernel Tuner provides developers with a number of optimization algorithms to choose from. For example, basin hopping, firefly, dual annealing and more. In addition to the pre-defined optimization algorithms, it also allows developers to wrap their own optimization strategy into Kernel Tuner, enabling them to integrate their own optimization algorithms [34].

2.4 Kernels

The kernels used in this research are 4 of the kernels that are part of the BAT [31] benchmarking suite. These kernels are described in detail in [31], but this section will also give a quick overview of the operation each kernel performs and the tunable parameters each kernel has.

2.4.1 GEMM

Generalized dense matrix-matrix multiplication (GEMM) is a kernel that executes matrix multiplication on two matrices, A and B:

$$C = \alpha A * B + \beta C$$

where C is the output matrix, and α and β are scalars [31].

The GEMM kernel used in this research is from CLBlast [15]. All tunable parameters can be found in Table 1

2.4.2 Convolution

The convolution kernel is used to extract features from an image, by going over the image with a filter. A 2D convolution on an image I with size $(w * h)$ and a filter F with size $(F_w * F_h)$ gives an output image O with size $((w - F_w) * (h - F_h))$:

Table 1: Tunable parameters for the GEMM kernel [12].

Parameters	Values
MWG	16, 32, 64, 128
NWG	16, 32, 64, 128
MDIMC	8, 16, 32
NDIMC	8, 16, 32
MDIMA	8, 16, 32
NDIMB	8, 16, 32
VWM	1, 2, 4, 8
VWN	1, 2, 4, 8
SA	0, 1
SB	0, 1

$$O(x,y) = \sum_{j=0}^{F_h} \sum_{i=0}^{F_w} I(x+i, y+j) * F(i,j)$$

The convolution kernel used in this research is the implementation from Van Werkhoven et al. [33], all tunable parameters for the convolution kernel are listed in Table 2 [31].

2.4.3 Dedispersion

Dedispersion is the process of reverting the dispersion of a radio signal that was transmitted through space over many frequencies (channels). Because different frequencies travel at different speeds, the radio signal is not received at once. The highest frequency f_h is received first, at time t_x. The lower frequencies arrive at time $t_x + k$, where k is the delay in seconds, as calculated by the dispersion equation:

$$k \approx 4150 * DM * (\frac{1}{f_i^2} * \frac{1}{f_h^2})$$

The input to the kernel are samples in time across all channels and outputs the dedispersed samples for many different dispersion measure DM values. To make this kernel parallelizable, each thread works on multiple samples and dispersion measures, while iterating over the channels [31]. All tunable parameters can be found in Table 2.

2.4.4 Hotspot

The Hotspot kernel solves a series of differential equations as part of an application that estimates processor temperature based on the processor architecture and power current that are simulated. The inputs to the kernel are the power and initial temperatures, the output is the average temperatue values spanning the chip, in grid format [31]. All tunable parameters can be found in Table 2.

Table 2: Tunable parameters for Convolution, Hotspot, and Dedispersion kernels. [12]

Parameter	Convolution	Hotspot	Dedispersion
block_size_x	$16k$ for k in $1, 2, \ldots, 16$	$1, 2, 4, 8, 16, 32k$ for k in $1, 2, \ldots, 32$	$1, 2, 4, 8, 16, 32$
block_size_y	$1, 2, 4, 8, 16$	$1, 2, 4, 8, 16, 32$	$8k$ for k in $4, 5, \ldots, 32$
tile_size_x	$1, 2, 3, 4$	$1, 2, 3, 4, 5, 6, 7, 8, 9, 10$	$1, 2, 3, 4$
tile_size_y	$1, 2, 3, 4$	$1, 2, 3, 4, 5, 6, 7, 8, 9, 10$	$1, 2, 3, 4, 5, 6, 7, 8$
read_only	$0, 1$	$0, 1$	
use_padding	$0, 1$		
use_shmem	$0, 1$		
temporal_tiling_factor		$1, 2, 3, 4, 5, 6, 7, 8, 9, 10$	
loop_unroll_factor_t		$1, 2, 3, 4, 5, 6, 7, 8, 9, 10$	
sh_power		$0, 1$	
tile_stride_x			$0, 1$
tile_stride_y			$0, 1$

These kernels were selected because fully brute-forced ache files are available from the Auto-TuningAssociation benchmark hub [36]. These cachefiles enable exact comparison against the global optimum for each kernel and architecture combinations without needing real time GPU executions.

Additionally, these BAT kernels represent real world applications from different domains, including image processing, linear algebra, astrophysics, and thermal simulation. Furthermore, they span a range of search space sizes, providing diversity in both domain and complexity.

2.5 Optimization Strategies

Inside Kernel Tuner [34] there are many optimization strategies that can be chosen from. In this research, the optimization strategies that will be used for comparison are the following: Basin Hopping, Genetic Algorithm, Differential Evolution, and Bayesian Optimization. This section will provide a concise explanation for each of these strategies.

2.5.1 Basin Hopping Strategy

The Basin Hopping strategy is based on the Basin Hopping algorithm. This algorithm repeats three steps. First it makes a small random change to the current position in the search space and does a local search from that position to find the nearest local minimum and accepts or rejects the new position. It is worth noting that a worse solution may still be accepted with some probability preventing the algorithm from acting strictly greedy [35]. This makes Basin Hopping well-suited for GPU kernel tuning, where the search space is expected to contain many local minima separated by large barriers [34].

2.5.2 Genetic Algorithm Strategy

The Genetic Algorithm strategy is based on the simple genetic algorithm [8, 34], which is based on biological evolution. The algorithm starts with a randomly generated population of n solutions, after which it follows 3 repeating steps from evolution: Selection, Crossover, Mutation. In the selection step, two solutions are chosen from the population, where solutions performing better have

a higher chance of being selected. The two selected solutions are then combined in the crossover step, where a random split point is chosen that decides which part of each selected solution is combined into the new solution. To avoid the selection and crossover step from keep producing the same solution, each parameter of the child's solution has a mutation probability of 0.1. The selection, crossover, and mutation steps are repeated until there are n new children. These n children replace the old population, after which the evolution process starts again.

2.5.3 Differential Evolution Strategy

The Differential Evolution strategy uses differential evolution [29]. Just like the genetic algorithm, this algorithm uses population-based methods and supports multiple methods for creating new members in the population. This algorithm is designed to find the global optimum in a continuous search space. This algorithm is capable of working with non-differentiable, nonlinear, and multimodal cost functions and is known to have consistent convergence, which makes it suitable for GPU auto-tuning. [34]

2.5.4 Bayesian Optimization Strategy

Bayesian Optimization consists of three components: an objective function, a surrogate model, and an acquisition function. The objective function is an unknown function for which we want to find the global optimum, for auto-tuning this is finding the best performing combination of tunable parameter values. The surrogate model approximates the unknown relationship between configurations and their execution times. This surrogate model is optimized by the acquisition function to suggest the next configuration to evaluate. Each evaluation updates the surrogate model, making the next suggestions more accurate [37].

2.6 Reinforcement Learning

Reinforcement Learning (RL) is a learning method in which an agent learns to make decisions by interacting with an environment. At each timestep, the agent observes the current state of the environment, takes an action based on this observation and gets back a reward signal, indicating how desirable this action was. By repeating this process, the agent learns to select actions that maximize the cumulative rewards over time. This learning process is visualized in Figure 2.

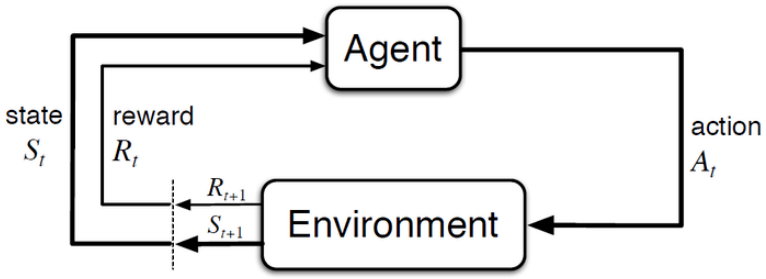


Figure 2: Agent environment loop [30].

Reinforcement learning is especially effective with sequential-decision problems, where the solution does not depend on a single decision, but on all decisions prior to the solution. To make a sequential-decision problem solvable by a reinforcement learning agent, we define a Markov Decision Problem (MDP) around this sequential-decision problem.

This section will further introduce the Markov Decision Problem (MDP) framework, followed by deep reinforcement learning, reward shaping, agent specific concepts relevant to PPO, and practical considerations relevant to training stable agents.

2.6.1 Markov Decision Problem

A MDP is a formal way of defining a sequential-decision problem and consists of the following 6 components:

1. **State Space $\mathcal{S}$:** The observations that are visible for the agent in a given state
2. **Action Space $\mathcal{A}$:** The actions the agent is able to take
3. **Transition Function $T(s'|s,a)$:** A conditional probability distribution defining how each taken action maps to a new state in the environment.
4. **Reward Function $r(s,a,s')$:** The function that decides the reward the agent gets for an action in a given state
5. **Discount Factor γ:** How much future rewards are valued, where $\gamma \in [0,1]$
6. **Initial State Distribution $p_0(s)$:** A distribution over the state space, stating where the agent starts in the environment.

These six components, combined with the Markov property, which states that each next state depends only on the current state and action taken, together define the MDP.

Given an MDP, the agent follows a policy $\pi : \mathcal{S} \to \mathcal{A}$, a mapping from each state to an action or a probability distribution over the actions. The goal of the agent is to maximize the cumulative discounted reward, which is called the return:

$$G_t = \sum_{i=0}^{\infty} \gamma^i r_{t+i}$$

To evaluate how good a state is under a given policy, the value function $V^\pi(s)$ is defined as the expected return when starting in state s and following policy π:

$$V^\pi(s) = E_\pi[G_t | s_t = s]$$

The core optimization problem that reinforcement learning aims to solve is finding a policy that maximizes the value function for all states in the state space, called the optimal policy:

$$\pi^* = argmax_\pi V^\pi(s)$$

2.6.2 Deep Reinforcement Learning

The MDP framework works well when the action and state spaces are small enough to represent them in an exact way. This exact way is called tabular reinforcement learning, where the exact value for each state or state-action pair can be collected in a table. But when problems start to become too large for exact tablular representations, which is the case for many real world problems, another representation is needed.

This is where deep learning becomes important, which uses neural networks to find approximations for large and complex environments [22]. When combining deep learning and reinforcement learning, an agent is created that can approximate an environment and has the potential to generalize across unseen states.

2.6.3 Reward Shaping

Defining the reward function is a critical part of reinforcement learning, since a misalligned reward function can cause the agent to optimize for a proxy objective, rather than the intended objective [18]. This motivates the use of potential-based reward shaping, which provides a principaled way of reshaping the reward while keeping its optimal policy.

Theorem 1 from Ng et al. (1999) [14] states that potential-based reward shaping is the only safe way of modifying the reward function, without risking a change in the optimal policy. The formal definition for a shaping function F as potential-based is the following:

$$F(s, a, s') = \gamma\phi(s') - \phi(s)$$

where F is the shaping function, $\phi(s')$ is a function over the state after the step, $\phi(s)$ is a function over the state before the step and γ is the discount factor.

The theorem also gives 2 conditions, either sufficient or necessity. The sufficient condition guarantees that if F is potential-based, all optimal policies under the original reward, stay optimal under the reward + F.

The necessity condition states that if F cannot be written as a potential-based function, there will always be an environment where F will cause the agent to find a different optimal policy than the original. This shows how incorrectly reshaping the reward can cause the agent to start optimizing something else than the goal.

When combining the theorem from Ng et al. [14] and the missalignment risk from Pan et al. [18] there is a clear reward design principle: the reward should directly reflect the true objective and extra shaping should always be potential-based to preserve the optimal policy.

2.6.4 Agent Specifics

Because of differences between Deep RL algorithms, some agent specific concepts used in the PPO [26] algorithm are shortly explained in this subsection, providing the needed background to understand the underlying mechanism of the trained agents.

On-Policy vs Off-Policy. A fundamental difference between reinforcement learning algorithms is whether they are on or off-policy. On-policy algorithms, like Proximal Policy Optimization (PPO) [26], generate episodes based on the policy that is being optimized. These algorithms do not keep track of earlier policy results after each policy update, so after each update new experiences have to be collected. Off-policy algorithms generate episodes using a different policy than the one being optimized.

Actor-Critic Methods. The core idea of the Actor-Critic method is that instead of only learning the policy, the agent also learns the value function. The actor is responsible for learning the policy, while the critic learns the value function. After the actor has taken an action, the critic will evaluate this action by estimating the value of the resulting state. This allows the agent to evaluate the quality of its actions immediately, instead of having to wait for the episode to end and receive the episode return. [30]

Proximal Policy Optimization. Proximal Policy Optimization (PPO) [26] is an on-policy, actor-critic reinforcement learning algorithm. This algorithm uses policy gradient methods to estimate the optimal policy, which can introduce problems where the agent makes very large policy updates at once, causing the policy to deteriorate. PPO addresses this problem by introducing a clipping factor, which constrains the size of policy updates. To measure the change between the new and old policy, PPO computes the probability ratio between them. A probability ratio close to 1 indicates a small policy update, and a ratio far from 1 indicates a large policy update. The clipping range is defined as follows: $[1-\epsilon, 1+\epsilon]$, where ϵ is usually set to 0.2 [26]. When performing the policy updates, both the clipped and unclipped policy gradient are calculated from which the minimum is used. The reason for using the minimum is that the advantage of an action can be both positive or negative, where the minimum operation has a different effect depending on the quality of the action. For actions with a positive advantage, the minimum selects the clipped objective once the ratio grows too large, preventing the policy from being overconfident about good actions. For actions with a negative advantage, the minimum selects the unclipped objective, ensuring that the full penalty is applied so the algortihm can still learn from bad actions. This makes the final objective a pessimistic bound, ensuring policy updates are conservative rather than optimistic [26].

2.6.5 Practical Considerations

When training Deep RL agents, there are practical considerations for healthy training behaviour. This section provides a concise explanation for the considerations used during the training of the agents in this research.

Input Normalization. Input normalization is the process of scaling observations to a bounded range. [24]. For on-policy deep reinforcement learning algorithms it is recommended to always use input normalization [2].

Time Awareness. A beneficial addition to the observation space is time awareness, Pardo et al. [20] show that a notion of time drastically improves the performance of PPO agents. Since the episode terminates after a fixed number of timesteps n, the MDP changes based on the current time step. Without adding time awareness to the agent's observation, it can not distinguise between 2 identical states that only differ by their remaining time. Without time awareness, the agent may learn suboptimal policies and exhibit instability due to incorrect credit assignment.

In this research, time is the number of steps taken within an episode, compared to the maximum allowed steps. This is included in the observation as step progress (Section 4.1.2), allowing the agent to separate a state where a configuration is reached early versus late in the tuning process.

Action Masking. When there are invalid states in an environment it is important to mask actions leading to these states to avoid wasted timesteps. Action masking solves this by creating a list of booleans for the current configuration, showing which actions can be taken and which ones can not. [9]

3 Related Work

This section provides an overview of relevant prior work in this research direction. It gives an overview of existing auto-tuning frameworks, how reinforcement learning is already being used for optimization and auto-tuning. The section ends with the current gap in the existing research that this work is based on.

3.1 Auto-Tuning Frameworks

Auto-Tuning GPU kernels is already well established, tools such as Kernel Tuner (KT) [34], CLTune [16], OpenTuner [3], Auto-Tuning Framework (ATF) [25],and Kernel Tuning Tools (KTT) [21] exist, which automatically tune GPU kernels written in CUDA [17] and OpenCL [27]. These tools use optimization algorithms to find a near optimal configuration in the search space, but none implement a reinforcement learning based strategy. One of the strategies implemented in KT is Bayesian Optimization [37], which is able to deal with the rough, discrete and constrained search spaces. Bayesian Optimization builds a surrogate model of the problem while optimizing, which helps the algorithm decide what configuration to try next. The fact that Bayesian Optimization can generalize well and consistently outperform other search strategies shows that there is value in using algorithms that learn from the search space. Where classic optimization algorithms and Bayesian Optimization are limited is the fact that these algorithms have no prior knowledge when they begin and cannot leverage experiences from previous tuning runs.

3.2 Reinforcement Learning for Combinatorial Optimization

Using Reinforcement Learning (RL) is a natural next step to leverage the experiences of previous tuning runs. Mazyavkina et al. [13] shows that RL models can become a promising direction for solving combinatorial optimization problems. This translates directly to auto-tuning GPU kernels, which is a combinatorial optimization problem.

3.3 Reinforcement Learning for Auto-Tuning

RL being applicable for auto-tuning problems can also be seen in other auto-tuning implementations. For example in compiler auto-tuning, where CompilerGym [6] exposes compiler optimization as an RL problem for an agent, Compiler-R1 [19] builds on that direction by using RL-trained LLMs to generate optimized compilers. Another implementation where RL has proven to be successful is automatic tuning of PID controllers [32], where the PPO [26] algorithm is used to tune the PID controller parameters.

More specifically, RL has already been applied to auto-tuning GPU kernels. Liu et al. [11] have created KernelGym, a GPU environment to train LLMs on creating optimal GPU kernels. They developed the Dr. Kernel model, which outperforms other LLMs on kernel generation. Similarly, AutoTriton [10] is another LLM based model, designed specifically for generating optimal Triton kernels.

3.4 Reinforcement Learning Optimization Strategy

The above approaches demonstrate that there has been a significant amount of research in GPU auto-tuning and RL methods for GPU auto-tuning. However, all existing RL based GPU auto-tuning approaches are LLM-based, no work has been done to implement an actual optimization strategy powered by RL, to the best of our knowledge. This research addresses this gap by developing an environment in which RL algorithms can be trained and deployed with Kernel Tuner to tune user-defined GPU kernels.

4 Approach

This section will provide an overview of the experiment approach and the implementation process, prior to the experiments. Two agents were developed, both sharing the same MDP formulation and environment structure, only differing in their action space. Figure 3 gives a big picture overview of the approach used.

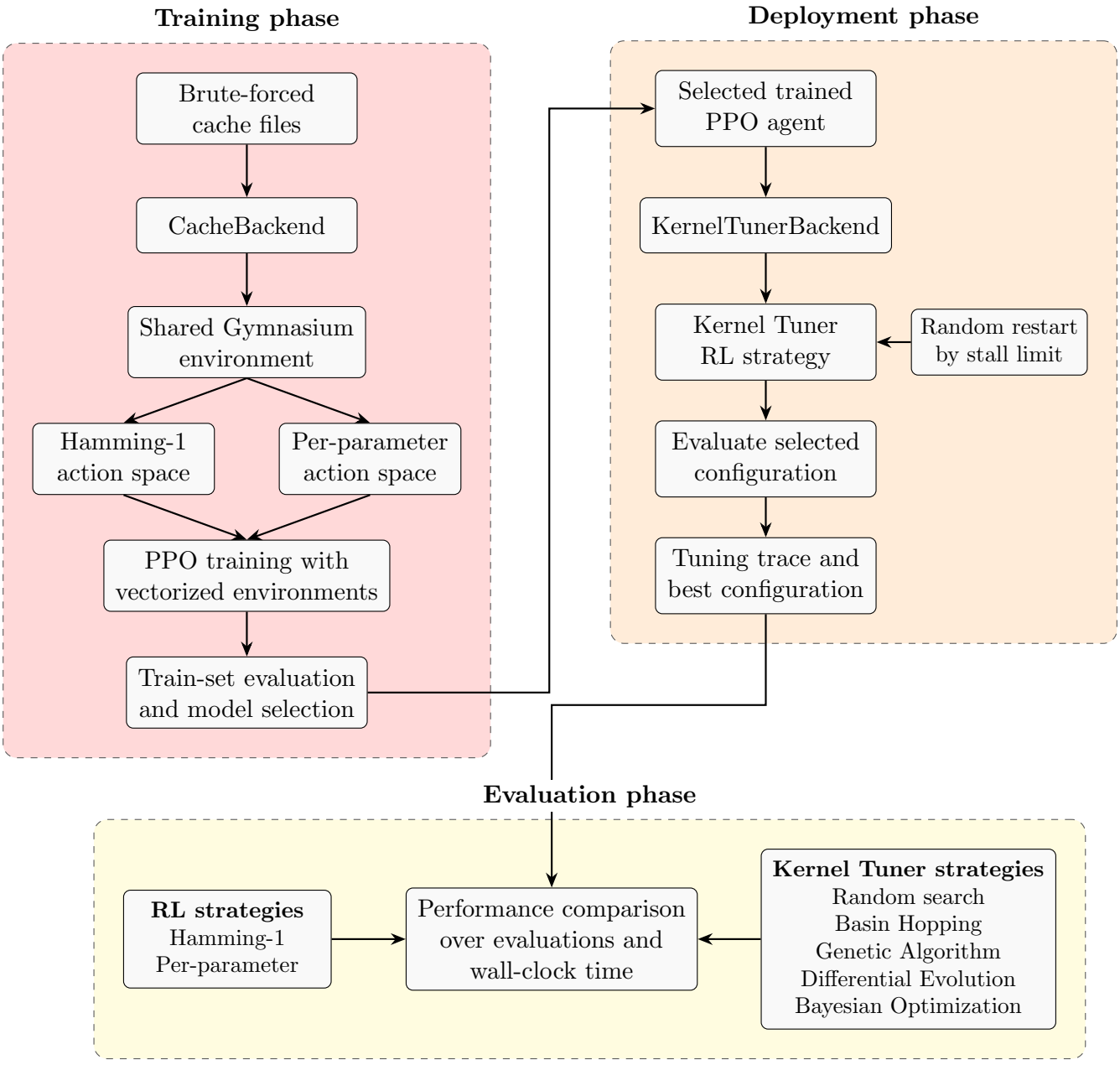


Figure 3: Overview of the approach used for the experiments in this work. Training uses brute-forced cache files through the CacheBackend and trains PPO agents with two different action spaces. During deployment the selected trained agent is used as a Kernel Tuner strategy through the KernelTunerBackend. The resulting agent based strategies are evaluated against existing Kernel Tuner optimization strategies.

4.1 Markov Decision Problem Definition

This section will explain the design of the MDP formulation used as a basis for the training environments. First it will go over the action space, explaining both the Hamming-1 and per-parameter action space, followed by the shared observation space and reward function.

4.1.1 Action

For the action space, two variants are developed. The first action space is based on Hamming-1, the second is based on per-parameter value selection. Both of the agents use action masking to avoid wasted actions.

Hamming-1 The first action space allows the agent to change exactly one parameter by a single step, either increasing or decreasing the chosen parameter by one option. This action space results in a size of $2 * parameterCount$, where parameterCount denotes the number of tunable parameters in the kernel.

Per-parameter Value Selection The second action space allows the agent more flexibility in choosing any value within a single parameter, instead of moving one step at a time. This allows the agent to make larger jumps between configuration, enabling faster traversal of the search space. The total size of the action space for this variant is the sum of the parameter options over each parameter.

Action Masking Both action spaces have a fixed size, but not all actions correspond to valid configurations. To handle these invalid actions, action masking is applied to filter invalid configurations from the search space as explained in Section 2.6.5.

4.1.2 Observation

The observation is defined as four components: the current kernel configuration, its execution time, the best seen execution time, and the step progress. The current configuration and its execution time inform the agent of its standing point in the search space, the best seen execution time shows the agent what it needs to improve on, the step progress informs the agent of its progress through the current episode, where one episode represents a single tuning run within the allowed tuning budget. The importance of this time awareness is explained in Section 2.6.5. All observations are normalized to $[0, 1]$ to stabilize training, following the recommendation for on-policy deep RL algorithms [2].

4.1.3 Reward

Properly defining and shaping the reward function is critical for an RL agent to successfully learn an optimal policy. For the reward function, potential-based reward shaping [14] is used. The reward function consists of two components: a base reward and a shaping reward. The agent gets the base reward of 1.0 when finding the optimal configuration, at which point the episode terminates. If the agent did not find the optimal configuration, the base reward is 0.0. In addition to the base reward, the agent receives a shaping reward, formally defined as $\gamma\phi(s') - \phi(s)$, where $\phi(s) = t_{opt}/t_{current}$. The base reward and shaping reward construct the total reward, which is formally defined as

$R = R_{true} + [\gamma\phi(s') - \phi(s)]$. The ratio of $\phi(s)$ will always be between $(0, 1]$, because t_{opt} is the smallest execution time. This creates a natural measure of progress towards the objective. A base reward of 1.0 is sufficiently large to have a meaningful impact on the total reward, since $\phi(s)$ is bounded in $(0, 1]$, meaning the shaping reward is bounded in $(-1, 1)$. A terminal reward of 1.0 therefore always makes a significant contribution to the total reward. Lastly, γ in the shaping function is the discount factor as used in the MDP, which is $\gamma = 0.999$. The motivation for the discount factor value is discussed in Section 4.3.2.

4.2 Environment

For the environment, Gymnasium is used [5]. Gymnasium is an open-source library providing a standard API for reinforcement learning environments. The main features of gymnasium are a set of environment abstractions, allowing wide compatibility between environments and training algorithms. This compatibility simplifies the development and testing of reinforcement learning algorithms. Gymnasium also provides a validation function that allows users to verify whether their custom environment meets the required interface specifications.

4.2.1 Environment Design

The environment used for the two agents consists of three classes, a base environment class, the Hamming-1 environment class, and the per-parameter environment class. The base environment class initializes the shared components used by both the Hamming-1 and per-parameter class. This base environment handles setting the environment attributes, normalizing the configuration and execution time, resetting the agent to a random position in the search space, and computing the reward after each timestep.

The Hamming-1 environment class uses the smaller Hamming-1 action space and the per-parameter environment class uses the bigger per-parameter value selection action space. Each subclass only implements the components that differ between the two action spaces, which are the action space definition, handling the action masking, and the step function. All three components ultimately result in the agent selecting a valid action and transitioning to the next state, but the internal mechanism by which the agent selects an action is different for both classes.

4.2.2 Backend Abstractions

To decouple the environment from the data source, this work implements two backends. This decoupling is necessary as the data source differs between training and deployment. During training the execution times are retrieved from cache files, but during deployment the agent gets the execution times directly from Kernel Tuner. These backends allow the environment to focus solely on the agent logic, independent of the underlying data source.

CacheBackend. This CacheBackend is used during the training phase of the agent. It retrieves all necessary data from the JSON cache file and stores it as class attributes. This allows the backend methods to work with the information from the cachefile directly.

KernelTunerBackend. The KernelTunerBackend is used for deployment. It works directly with the Searchspace class used by Kernel Tuner (KT). This ensures the agent can be deployed as a KT

strategy, as the strategy function works with the KT Searchspace class. During deployment, there are no pre-computed stats available, so the backend estimates normalization values from a set of random samples taken during initialization of the backend. The number of random samples is set to five by default. This estimation introduces a change in distribution compared to training, where exact statistics were available from cache files and may cause a variance in the performance during deployment. The effect of different sample sizes is discussed in Section 5.3.3.

4.2.3 Multiple Environment Setup

The agent is trained on multiple GPU-specific searchspaces simultaneously. This encourages the agent to find a policy that generalises across different GPU architectures, rather than overfitting to a single one. Stable-Baselines3 [23] provides vectorised environments, allowing the agent to update its policy based on experiences from multiple search spaces simultaneously. Two vectorised environment implementations are available: DummyVecEnv and SubprocVecEnv. Both work for the same environments, the difference is that SubprocVecEnv spawns a subprocess for each search space, where DummyVecEnv steps through all environments sequentially. DummyVecEnv is used for training, as the environment step consists of a simple cache lookup rather than a computationally expensive operation. Using SubprocVecEnv would introduce unnecessary parallelism overhead, reducing overall training efficiency.

4.3 Agent Choice

This work applies the PPO [26] algorithm, selected based on the properties of the MDP following the framework of Bongratz et al. [4]. The choice of a model-free, on-policy, actor-critic algorithm is motivated by three properties of the problem. First, kernel execution times can only be determined by running the actual kernel, ruling out model-based approaches, since model-based approaches would need to predict how the environment behaves without running the kernel. Second, the combination of a neural network and bootstrapping makes on-policy training necessary to avoid the training instability caused by the deadly triad [30]. Using an on-policy avoids this deadly triad by ensuring the agent only learns from actions chosen by its current policy. Third, an actor-critic architecture is preferred over pure policy-based methods, as the critic reduces gradient variance and improves sample efficiency, while pure value-based methods are not designed for direct policy optimization in large discrete spaces. Among on-policy actor-critic algorithms, PPO is specifically chosen for its clipped surrogate objective and entropy regularisation, both of which contribute to training stability. This choice is further supported by CompilerGym [6], which demonstrates the applicability of PPO on a similar combinatorial optimization problem with a discrete action space.

4.3.1 Agent Training Procedure

Both agents are trained via Stable-Baselines3 on multiple environments simultaneously. For Convolution and Dedispersion, the agents were allocated 600K timesteps. As the search spaces of GEMM and Hotspot are considerably larger than those of Convolution and Dedispersion, the agents were allocated a training budget of 2M timesteps. The timestep budgets for all four kernels were determined based on training curve convergence.

4.3.2 Hyperparameter Configuration

The default PPO hyperparameters from Stable-Baselines3 were used as a starting point. Two hyperparameters were adjusted: the entropy coefficient and the discount factor. The entropy coefficient was tuned from 0.00 to 0.01, to stimulate exploration more and lower the chance of getting stuck in a local minimum. The discount factor was tuned from 0.99 to 0.999, since episode length is capped at 1000 timesteps and the agent should reach the optimal configuration within this budget. By using a higher discount factor, the terminal reward remains more significant through the whole episode, stimulating the agent more to find this configuration.

As part of this research, a hyperparameter search using Optuna [1] was conducted on the Convolution and Hotspot kernels to investigate whether a better shared configuration could be found. This search revealed that optimal hyperparameter configurations differ significantly between the kernels. This suggests that a single shared hyperparameter configuration would be a compromise between all kernels, since no configuration is optimal for all kernels at the same time. Tuning a model specifically per kernel could improve performance per kernel, but finding this general configuration across all kernels is a complex problem, left for future work.

4.3.3 Agent Model Selection

Per kernel, 8 different models are trained. To select the best performing model out of these, they are evaluated on the train-sets, from which the best performing model is chosen. The selection metric is the average gap between the best found configuration and the optimal configuration, normalized by the distance between the median and optimal execution time. A score of 0.00% means the agent consistently found the optimal configuration, a score higher than 100.00% means it performed worse than median. The evaluation is averaged over 20 episodes per GPU. The deployment GPUs are held out from the evaluation, ensuring the selection is done purely on the train-set. The performance of all models and the average over the models per kernel can be found in appendix B. The model used is highlighted in the appendix. These tables also reveal seed sensitivity in training, in particular for the per-parameter agent on GEMM and for the Hamming-1 agent on Dedispersion.

4.4 Kernel Tuner Integration

To enable KT to use the agents as an optimization strategy, a strategy file is created, exposing a `tune()` entry point via which KT can call the optimization strategy. Inside the entry point the agent gets loaded, if the kernel being tuned is not one of the four the agents were trained on, the strategy raises an error and terminates. During the tuning time, the agent traverses the search space, greedily selecting the best action at each step. The agent acts greedily, because `deterministic=True` is set on deployment, meaning pure greedy exploitation during deployment, which differs from the stochastic policy used during training. To avoid the agent getting stuck and wasting its tuning budget, a stall limit is implemented. When the agent does not improve the current configuration within N steps, the agent is reset to a random place in the environment, preventing the agent from getting stuck in local minima. The decision to add this stall limit was made, because with an earlier version without a stall limit, the agent would improve in the beginning of the tuning, but stop improving completely after this initial start. Tuning continues until KT stops the tune function by throwing a StopCriterionReached exception. On termination,

the execution times and configurations collected by CostFunc are returned to KT as the strategy results.

4.5 Experiment Methodology

For the experiments that will be used to draw the conclusions from, the methodology from Willemsen et al. (2024) [36] will be used, which serves as a unified methodology for comparisons between optimization algorithms in auto-tuning.

This methodology consists of 4 main steps: Experimental Setup, Optimization Algorithms, Tuning Budget and Noise, and Reporting Optimization Algorithm Performance. These steps will be explained in detail below.

The experiments are conducted using the software package, developed by Willemsen et al., following this methodology and handles the execution, data collection, and visualisation.

4.5.1 Experimental Setup

The necessary parts of the experimental setup are that an established benchmark suite is used and that the searchspace is left as is. This makes sure the searchspace is valid and not optimized for a particular optimization strategy.

The tests should be done on different kernels and GPU architectures to ensure the results are generalized and not tailored to single combinations of GPU and kernel. The kernels used are: Convolution, dedispersion, GEMM and Hotspot, as explained in Section 2.4.

To ensure the experiment meets these requirements, pre-computed cache files from the Auto-TuningAssociation's benchmark hub are used. The benchmark hub is a FAIR-compliant hub for GPU auto-tuning benchmarking resources, which provides fully brute-forced search spaces together with their source files. The sources of the kernels are CLBlast [15] for the GEMM kernel and BAT [31] for the Convolution, dedispersion, and Hotspot. These kernels have been searched with the brute-force algorithm on these GPU architectures: NVIDIA A100, NVIDIA A4000, NVIDIA A6000, AMD W6600, and AMD W7800 of DAS-6 and the AMD MI250X of LUMI. The combination of these kernels and GPU architectures has created a diverse set of searchspaces to experiment with. The set is diverse because it spans 2 different vendors: NVIDIA and AMD, and 2 hardware grades: professional workstation grade (A4000, A6000, W6600, W7800) and data center grade (A100, MI250X).

Using these cache files instead of actual GPU evaluations has 2 main advantages. Firstly, it ensures that all optimization strategies are compared on the same data by removing other hardware related variability. Secondly, it increases the efficiency of the research by bypassing the benchmark time Kernel Tuner needs on each kernel configuration.

4.5.2 Optimization Algorithms

To evaluate the performance of the optimization algorithm, we need to compare it against other strategies. These strategies that the reinforcement learning agent will be evaluated against come from kernel tuner version 1.3.1 and are: random search, which picks a kernel configuration from the searchspace at random, Basin Hopping, Genetic Algorithm, Differential Evolution, and Bayesian Optimization. These optimization strategies are explained in Section 2.5. By using these, the evaluation consists of a baseline algorithm and state-of-the-art optimization algorithms.

The strategies that will be compared against the existing strategies are the reinforcement learning agents as described in Section 4.3, trained in the environments described in Section 4.2.

4.5.3 Tuning Budget and Noise

Deciding the tuning budget makes sure all optimization algorithms have the same room for optimization. To decide this budget, we take the budget that random search needs, to get within 2.5% of the distance between the searchspace median and the optimum. This means that there will be a separate budget for each different kernel searchspace. This budget will be expressed both in function evaluations and wall-clock time.

To limit the noise, each kernel parameter configuration chosen by each optimization algorithm will be sampled at least 30 times and each experiment will be repeated for 100 iterations. The results will average over these repetitions to get a reliable result back from the experiments.

4.5.4 Reporting Optimization Algorithm Performance

To report the performance, the random search performance will be used as a baseline. For the optimization algorithms a visualization will be made over function evaluations and over wall-clock time. To compare across searchspaces, the results will be normalized and shown in a single plot.

5 Experiments

This section presents the evaluation results from the experiments run, as explained in Section 4.5. First, the aggregated results over all kernels will be presented. After which, the kernel specific aggregate results are presented. All graphs are normalized around the baseline (random search), which shows the improvement of the algorithms compared to this baseline. All per kernel and GPU results can be found in Appendix A, which present the tuning curves over all function evaluations and over wall-clock time.

5.1 Search Strategy Comparisons

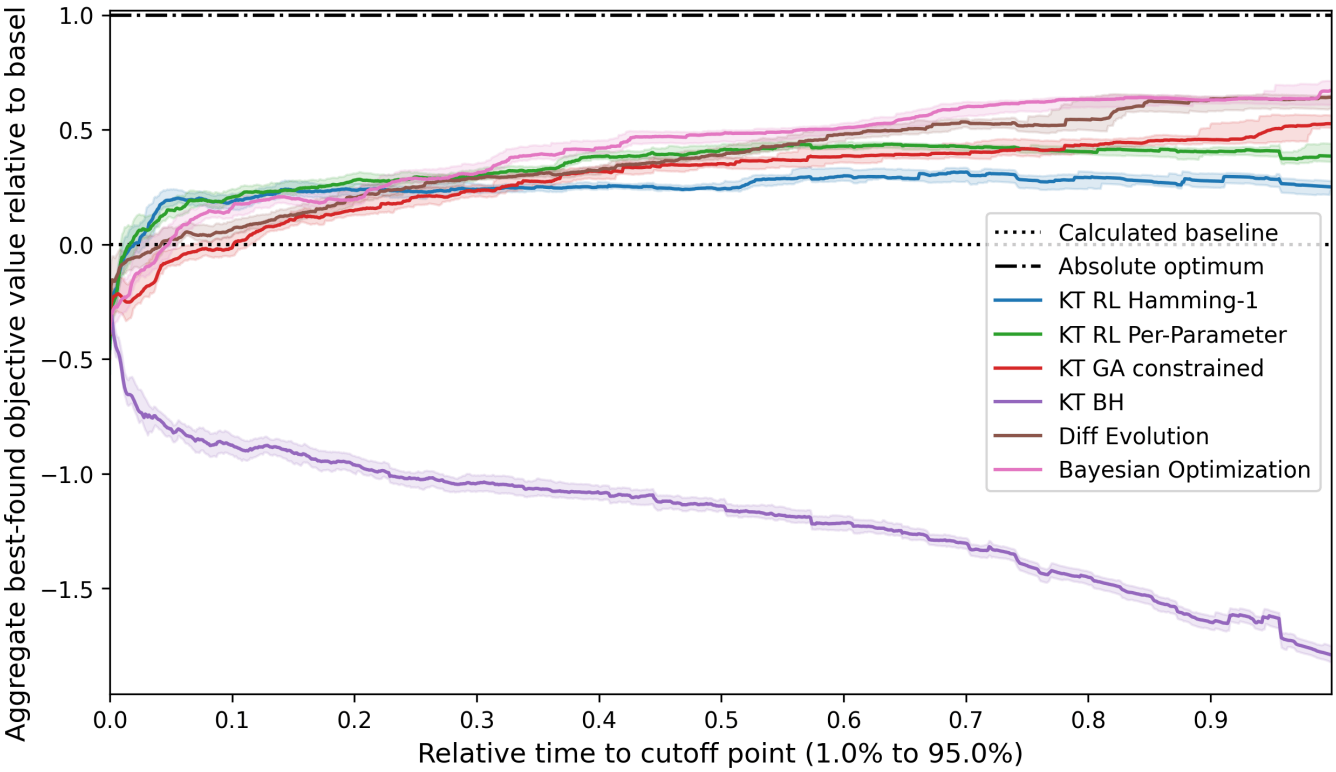
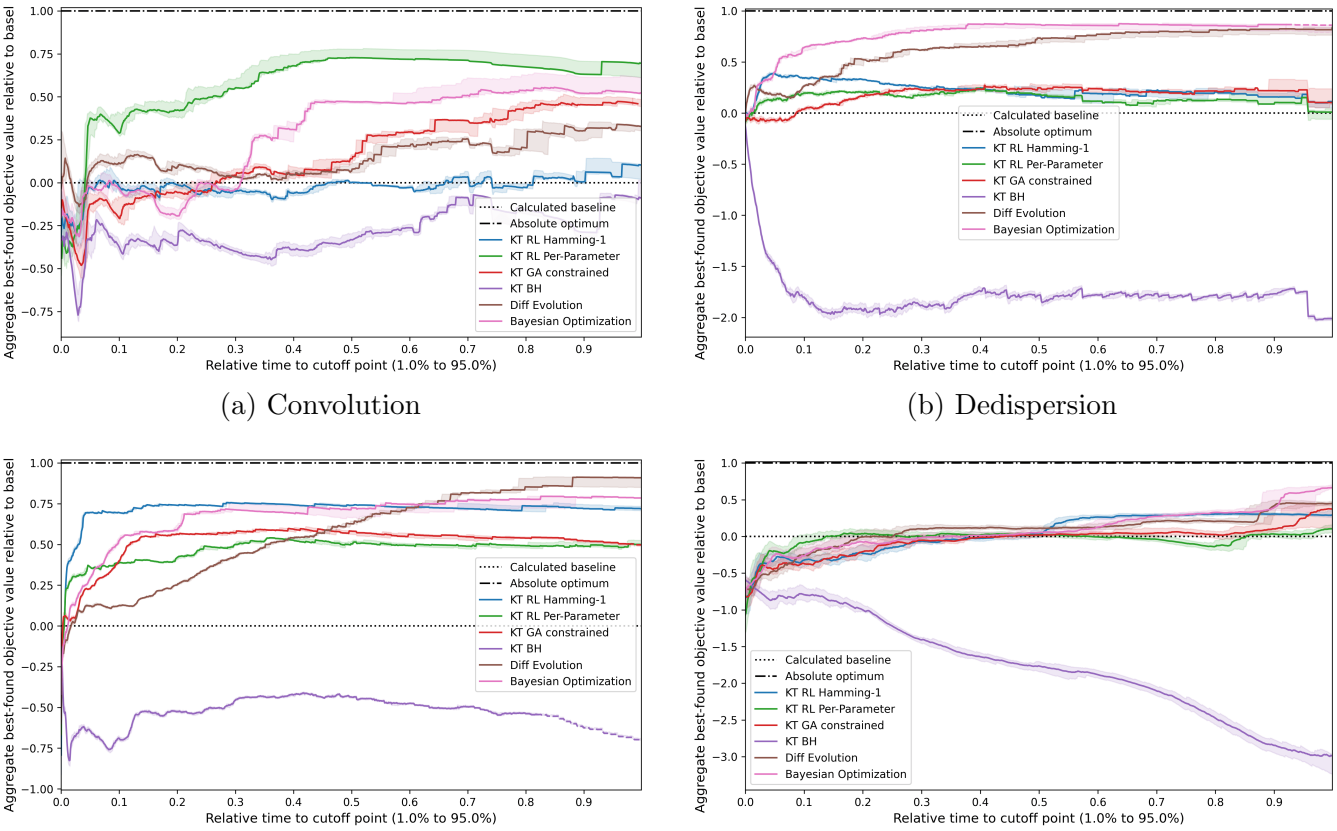


Figure 4: Aggregated performance over all kernels on A6000 and W7800



(a) Convolution

(b) Dedispersion

(c) GEMM

(d) Hotspot

Figure 5: Aggregated performance per kernel on A6000 and W7800

Figure 4 presents the aggregated performance of all optimization algorithms across all four kernels. At the end of the tuning budget, Bayesian Optimization (BO) performs best, with Differential Evolution (DE) performing as a close second. Genetic Algorithm (GA) starts of lower than all optimization strategies except Basin Hopping (BH), but ends up overtaking both agents and ends as third best overall. GA is followed by the per-parameter agents and the Hamming-1 agent respectively. BH performs worst overall, remaining well below random search throughout the entire tuning budget.

Performance differences between algorithms are most notable in the early stages of tuning. In the first quarter of the tuning budget, the per-parameter agent performs best, followed by BO and the Hamming-1 agent. Beyond 30% of the tuning budget, BO overtakes the per-parameter agent and remains above all other algorithms. Beyond 60%, DE also overtakes the per-parameter agent. Lastly, beyond 80%, GA overtakes the per-parameter agent.

The early strong performance of the RL agents suggests that the learned policy provides a good starting point, but the agents eventually get overtaken by algorithms that continue to improve throughout the tuning budget.

5.1.1 Convolution Results

Figure 5a presents the performance of the optimization algorithms on the Convolution kernel. At the end of the tuning budget the per-parameter agent performs best, followed by BO, GA and DE

respectively. The Hamming-1 agent ends slightly above random. BH performs worst overall, ending below the baseline.

A notable difference exists between the performance of the per-parameter agent and the Hamming-1 agent. The per-performance agent shows a sharp improvement early in the tuning budget and ends as best overall, while the Hamming-1 agent acts comparable to random search through the entire tuning budget.

The per-parameter agent's ability to select any value from a single parameter likely enables more effective exploration of the convolution search space, compared to the incremental approach of the Hamming-1 agent.

5.1.2 Dedispersion Results

Figure 5b presents the performance of the optimization algorithms on the Dedispersion kernel. The performance of the algorithms is consistent with the aggregated results.

Both BO and DE perform the best overall, with both approaching the optimum by the end of the tuning budget. The two agents and GA follow a similar trajectory over the tuning budget, with the per-parameter agent ending slightly below GA and the Hamming-1 agent. However, in the beginning both agents perform better than GA, with the Hamming-1 agent briefly outperforming GA, the per-parameter agent, and DE. BH performs well below random search on this kernel.

The similar trajectories of both agents with GA throughout the tuning budget show that both agents perform comparably to GA.

5.1.3 GEMM Results

Figure 5c presents the performance of all optimization algorithms on the GEMM kernel. At the end of the tuning budget, DE performs best, ending near the optimum, with BO performing comparably as a close second. Both agents follow a similar path in the second half of the tuning budget, ending slightly below BO.

The difference in performance is strongest in the early stages of tuning. Both agents, BO, and GA shows a steep improvement early in the tuning budget. The Hamming-1 agent has the largest early improvement of all optimization algorithms, after which it plateaus and improves only slightly. The per-parameter follows the same trajectory, but starts with a smaller early improvement and increases more gradually than the Hamming-1 agent. DE improves more slowly than the other algorithms, but keeps improving throughout the whole tuning budget, which results in approaching the optimal solution near the end. BH performs well below the baseline.

Both agents achieve most of their improvement early and show limited improvement during the remainder of the tuning budget, while Differential Evolution and Bayesian Optimization continue improving.

5.1.4 Hotspot Results

Figure 5d presents the performance of all optimization algorithms on the Hotspot kernel. At the end of the tuning budget, BO performs best, with DE and GA performing as a close second and third. The Hamming-1 agent ends slightly below GA, followed by the per-parameter agent, which ends just above the baseline.

All algorithms, except for BH, are gradually improving through the whole tuning budget. The per-parameter makes the biggest improvement in the early stage of the budget, but plateaus after the initial improvement and gets overtaken by most of the other algorithms, but near the end starts to improve again. The Hamming-1 agent starts of slower than the per-parameter agent, but starts to improve around the middle of the tuning budget, overtaking all algorithms for a moment. BH performs below the baseline.

Testing on the Hotspot kernel with time as the performance objective resulted in a relatively small amount of function evaluations, with a maximum of 30 function evaluations. An alternative tuning objective using GFLOP/s was also explored to address this. The resulting plots are included in Appendix A.1.

5.2 Agent Comparison

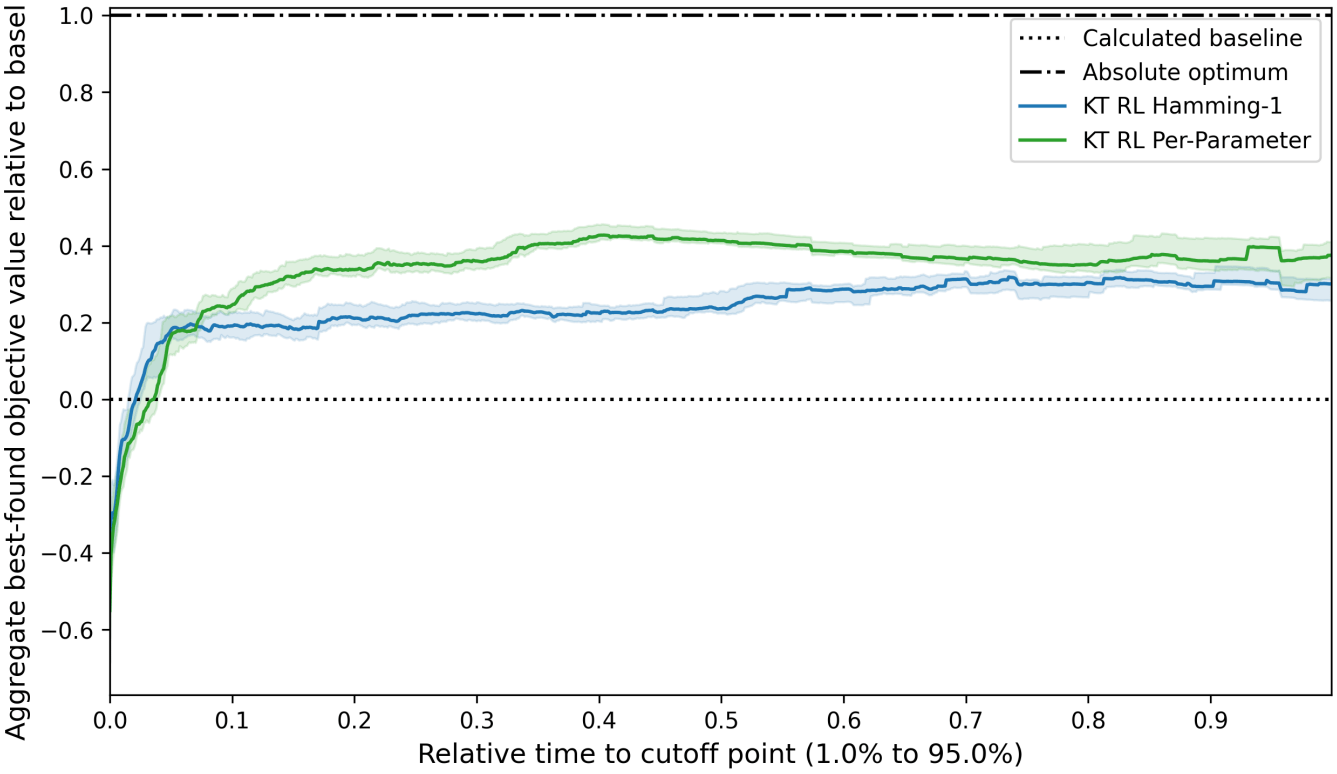


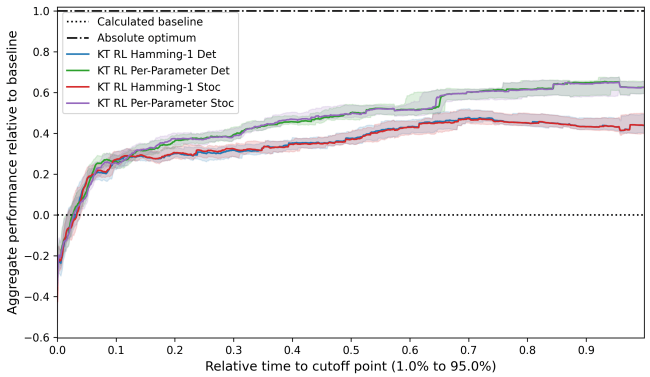
Figure 6: Aggregated performance over all kernels on A6000 and W7800

Figure 6 presents the aggregated performance of the two agents across all four kernels. At the end of the tuning budget, the per-parameter agent ends closer to the optimum than the Hamming-1 agent.

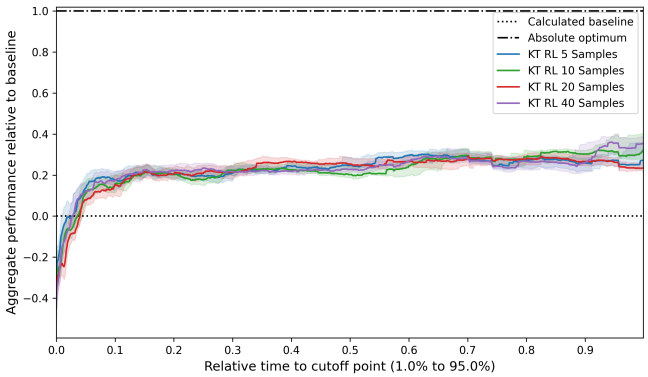
A notable similarity between the agents is the overall trajectory. Both agents make the largest improvement in the early stage of the tuning budget, after which they both plateau. The per-parameter agent manages to improve more than the Hamming-1 agent early on, enabling the per-parameter agent to find a better configuration by the end of the tuning budget.

Both agents achieve their largest aggregate performance gains during the early stage of tuning and show limited improvement during the later stage.

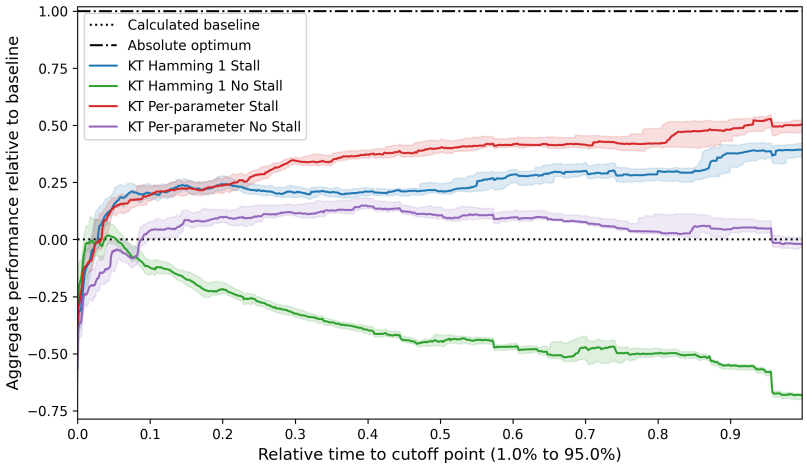
5.3 Agent Behavior Analysis



(a) Aggregated performance over all kernels on A6000 and W7800, comparing a deterministic strategy and a stochastic strategy for both agents.



(b) Aggregated performance over all kernels on A6000 and W7800, comparing four different estimation sample sizes: 5, 10, 20, and 40



(c) Aggregated performance over all kernels on A6000 and W7800, comparing both the Hamming-1 and per-parameter agents with and without a stall limit reset mechanism.

Figure 7: Deployment analysis for the agents on A6000 and W7800.

		Mean jump distance		Unique configurations seen		Revisit percentage (%)	
Kernel	GPU	H1	PP	H1	PP	H1	PP
Convolution	A6000	1.20	3.49	335	235	76.7	89.9
Convolution	W7800	1.30	2.27	35	28	45.3	79.1
Dedispersion	A6000	1.54	4.01	381	328	73.3	84.7
Dedispersion	W7800	1.49	3.18	1055	868	83.5	89.5
GEMM	A6000	1.05	1.18	1528	1485	78.0	80.2
GEMM	W7800	1.06	1.14	1662	1510	78.8	78.9
Hotspot	A6000	1.00	1.48	17	14	15.0	50.0
Hotspot	W7800	1.19	2.15	34	30	49.3	54.5

Table 3: Trace statistics for the Hamming-1 (H1) and per-parameter (PP) agents.

To examine the behavior of the agents during deployment, the tuning traces are analyzed using four metrics: the mean jump distance, the number of unique configurations seen during the tuning, revisit percentage, and the percentage of evaluations used on the two most visited configurations. Mean jump distance measures the mean index distance moved within a changed parameter. Unique configurations is the amount of different configurations seen by the agent during tuning. Revisit percentage is the share of revisited configurations over the whole tuning budget. Table 3 presents these metrics for both agents on each kernel and GPU.

5.3.1 Trace Analysis

The per-parameter agent has a larger mean jump distance for every kernel and GPU combination. This confirms that its action space provides greater mobility for a single step. The mean jump includes both policy actions and random restarts caused by the stall limit. Since restarts can change multiple parameters at once, the mean jump gap is larger than the action space would normally allow. Since both agents use the same restart mechanism, the metric remains usefull for comparing the mobility between both agents.

This greater mobility does not result in a higher search space coverage. The per parameter agent visits less unique configurations than the Hamming-1 agent does. It also has a higher revisit percentage on all kernel and GPU combinations. The results show that the per-parameter agent has a greater search space mobility, but it does not result in more exploration.

The traces also show that the agents use a large part of their given tuning budget on already seen configurations rather than exploring unseen ones. Table 3 shows that the revisit percentages exceed 45% on most kernel and GPU combinations for both agents, with a highest revisit rate of 89.9%. This behavior is consistent for both agents, indicating that when a high-performance region is found, the agents repeatedly re-evaluate a small set of configurations inside this region, instead of continuing to explore.

5.3.2 Deterministic versus Stochastic

The main experiment used a deterministic deployment, where the action with the highest probability in the policy is selected. To test whether this choice was the cause of the plateau, since deterministic deployment does not explore based on the policy probabilities, the agents were also deployed stochastically. During the stochastic deployment, the agents sample from the policy probability, instead of choosing the highest probability. Figure 7a shows that deterministic and stochastic deployment follow an almost identical tuning trace. Deterministic action selection is therefore not the main cause of the plateaus, but indicated that the learned policy assigns most of its probability to the same actions.

5.3.3 Number of Estimation Samples

Figure 7b compares deployment using 5, 10, 20, and 40 samples to estimate the normalization. The resulting tuning curves are almost identical, which shows that increasing the number of samples does not increase performance of the agents. However, all settings still rely on estimated statistics, so a difference between estimated and cache file statistics normalization still remains a gap between training and deployment.

5.3.4 Stall Limit

The repetitive behavior is consistent with the plateaus observed during tuning. During testing, deployment without a stall limit showed the agent stopping improvement shortly after the initial performance gain. Figure 7c compares the overall performance with and without a stall limit, confirming that the stall limit has a possitive effect on overall performance. With the stall limit, random restarts reset the agent to a random config after a number of steps without improvement, enabling further improvement in the later stages of the tuning budget. This shows that the learned policy quickly reaches a high performing region, but struggles to leave or refine the region without the stall limit resets.

6 Conclusions and Further Research

This thesis investigated how PPO-based agents with different action spaces perform compared to existing optimization strategies in Kernel Tuner. The results show that both agents outperform random search on most kernels and can be a competitive optimization strategy on the Convulution and GEMM kernels. The agents are especially competitive in the early stage of the tuning budget, where they make the largest improvement on most kernels. However, the early improvement is not enough to outperform more consistent strategies like Bayesian Optimization and Differential Evolution.

When looking at the agents' performance, the results show that the two agents can perform similarly or very differently across kernels. On the Convolution kernel the per-parameter agent performs better than the Hamming-1 agent, but on the Hotspot kernel, the Hamming-1 agent performs better. However, on the GEMM and Dedispersion kernels, both agents perform comparably.

The trace analysis shows that the per-parameter action space provides greater search mobiility, but does not explore a larger part of the search space, since both agents frequently revisit the same configurations. Implementing a stall limit allowed the agents to slightly improve during the later stages, but pointed out that the learned policy quickly locates a high performing region, but do not have the ability to refine further without random restarts.

In conclusion, a reinforcement learning approach of two PPO-based agents with different action spaces shows promising results in the early stages of the tuning budget, but cannot compete with the continuous improvement throughout the whole tuning budget of other optimization strategies. This approach is most valuable when agents can be trained once and deployed across many new GPU architectures, paying off the training cost over time.

6.1 Limitations

Some limitations of this work are the following. First, the current approach required brute-forced data to train, which is expensive to obtain. If brute-forced cache files are not available, KernelTunerBackend could be used for live training, but would be significantly slower and computationally expensive. Second, during deployment, precomputed search space statistics are unavailable, so the KernelTunerBackend estimates normalization values from 5 random evaluations at episode start. This introduces a distribution shift from training, where exact statistics were available from cache files, and may contribute to variance in deployment performance. Third, the agents used one set of hyperparameters across kernels instead of tuning hyperparameters per kernel. Fourth, the agents plateau after their initial performance gain and rely on random restarts to improve in the later stages of the tuning budget, unlike Bayesian Optimization and Differential Evolution. Lastly, for the Hotspot kernel the evaluation resulted in relatively few evaluations. To address this an alternative tuning objective using GFLOP/s was suggested, but since the agents are trained on execution time, using the GFLOP/s as an accurate tuning objectvie would require retraining with a different reward signal, which was not feasable in the time span of this project.

6.2 Further Research

Several directions for future work are identified. First, an Optuna search showed significant differences between the best performing agent hyperparameters across kernels. Tuning a single model for each kernel or investigating further if a better shared configuration exists could improve agent performance.

Second, methods to allow the agent to leave high performing regions in the search space without random restarts should be investigated. Possible approaches could be masking recently visited configurations, adding information about the recently visited configurations to the observation space, detecting repeated configuration cycles, or changing the training to encourage more exploration.

Third, the agent's ability to generalize across unseen GPUs could be improved by enhancing the observation space with GPU specific features. By enabling the agent to take the underlying hardware into account, it may learn to recognize patterns that transfer better across different GPU architectures. Investigating which GPU features are most influential on the agent performance remains an open research question, but will require feature extraction from GPU hardware specifications at deployment time, which is currently not supported by the reinforcement learning strategies.

Lastly, a Model-Agnostic Meta Learning (MAML) [7] approach is a promising direction for future research. MAML trains a model to find initial weights that can quickly adapt to new tasks with only a few gradient steps. This could potentially enable an agent to generalize to unseen kernels and GPU architectures with minimal additional training, removing the need for a separate model per kernel as in the current approach. However, some challenges need to be addressed for this approach. First, the observation space size differs between kernels, which causes issues when using neural networks, since these require a fixed input size. Second, MAML uses second-order gradients, which scale as n^2 with the number of network parameters, making the training significantly more expensive than standard PPO. A first-order approximation (FOMAML) exists that avoids second-order gradients, significantly reducing computational cost at the expense of some gradient information [7]. Despite these challenges, MAML or FOMAML remain interesting next research directions for improving generalization across kernels and GPU architectures.

References

[1] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework, 2019.

[2] Marcin Andrychowicz, Anton Raichuk, Piotr Stańczyk, Manu Orsini, Sertan Girgin, Raphael Marinier, Léonard Hussenot, Matthieu Geist, Olivier Pietquin, Marcin Michalski, Sylvain Gelly, and Olivier Bachem. What matters in on-policy reinforcement learning? a large-scale empirical study, 06 2020.

[3] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. Opentuner: An extensible framework for program autotuning. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Edmonton, Canada, Aug 2014.

[4] Fabian Bongratz, Vladimir Golkov, Lukas Mautner, Luca Della Libera, Frederik Heetmeyer, Felix Czaja, Julian Rodemann, and Daniel Cremers. How to choose a reinforcement-learning algorithm, 2024.

[5] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[6] Chris Cummins, Bram Wasti, Jiadong Guo, Brandon Cui, Jason Ansel, Sahir Gomez, Somya Jain, Jia Liu, Olivier Teytaud, Benoit Steiner, Yuandong Tian, and Hugh Leather. Compilergym: Robust, performant compiler optimization environments for ai research, 2021.

[7] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks, 2017.

[8] John H. Holland. Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence. 1992.

[9] Shengyi Huang and Santiago Ontañón. A closer look at invalid action masking in policy gradient algorithms. *The International FLAIRS Conference Proceedings*, 35, May 2022.

[10] Shangzhan Li, Zefan Wang, Ye He, Yuxuan Li, Qi Shi, Jianling Li, Yonggang Hu, Wanxiang Che, Xu Han, Zhiyuan Liu, and Maosong Sun. Autotriton: Automatic triton programming with reinforcement learning in llms, 2025.

[11] Wei Liu, Jiawei Xu, Yingru Li, Longtao Zheng, Tianjian Li, Qian Liu, and Junxian He. Dr. kernel: Reinforcement learning done right for triton kernel generations, 2026.

[12] Milo Lurati, Stijn Heldens, Alessio Sclocco, and Ben van Werkhoven. Bringing auto-tuning to hip: Analysis of tuning impact and difficulty on amd and nvidia gpus. In Jesus Carretero, Javier Garcia-Blas, Sameer Shende, Ivona Brandic, Katzalin Olcoz, and Martin Schreiber, editors, *Euro-Par 2024: Parallel Processing*, pages 91–106, Germany, 2024. Springer Science and Business Media Deutschland GmbH.

[13] Nina Mazyavkina, Sergey Sviridov, Sergei Ivanov, and Evgeny Burnaev. Reinforcement learning for combinatorial optimization: A survey, 2020.

[14] Andrew Y. Ng, Daishi Harada, and Stuart J. Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *Proceedings of the Sixteenth International Conference on Machine Learning*, ICML '99, page 278–287, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[15] Cedric Nugteren. Clblast: A tuned opencl blas library. In *Proceedings of the International Workshop on OpenCL*, IWOCL '18, page 1–10. ACM, May 2018.

[16] Cedric Nugteren and Valeriu Codreanu. Cltune: A generic auto-tuner for opencl kernels. In *2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip*, page 195–202. IEEE, 2015.

[17] NVIDIA Corporation. *CUDA C++ Programming Guide*, 2026.

[18] Alexander Pan, Kush Bhatia, and Jacob Steinhardt. The effects of reward misspecification: Mapping and mitigating misaligned models, 2022.

[19] Haolin Pan, Hongyu Lin, Haoran Luo, Yang Liu, Kaichun Yao, Libo Zhang, Mingjie Xing, and Yanjun Wu. Compiler-r1: Towards agentic compiler auto-tuning with reinforcement learning, 2025.

[20] Fabio Pardo, Arash Tavakoli, Vitaly Levdik, and Petar Kormushev. Time limits in reinforcement learning, 2022.

[21] Filip Petrovič, David Střelák, Jana Hozzová, Jaroslav Ol'ha, Richard Trembecký, Siegfried Benkner, and Jiří Filipovič. A benchmark set of highly-efficient cuda and opencl kernels and its dynamic autotuning with kernel tuning toolkit. *Future Generation Computer Systems*, 108:161–177, 2020.

[22] Aske Plaat. *Deep Reinforcement Learning*. Springer Nature Singapore, 2022.

[23] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8, 2021.

[24] Nirnai Rao, Elie Aljalbout, Axel Sauer, and Sami Haddadin. How to make deep rl work in practice, 2020.

[25] Ari Rasch, Michael Haidl, and Sergei Gorlatch. Atf: A generic auto-tuning framework. In *2017 IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 64–71, 2017.

[26] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.

[27] Guochun Shi, David Gohara, and John E. Stone. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems . *Computing in Science & Engineering*, 12(03):66–73, May 2010.

[28] David Silver. Lectures on reinforcement learning. URL: https://www.davidsilver.uk/teaching/, 2015.

[29] Rainer Storn and Kenneth Price. Differential evolution – a simple and efficient heuristic for global optimization over continuous spaces. *J. of Global Optimization*, 11(4):341–359, December 1997.

[30] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.

[31] Jacob O. Tørring, Ben van Werkhoven, Filip Petrovč, Floris-Jan Willemsen, Jiří Filipovič, and Anne C. Elster. Towards a benchmarking suite for kernel tuners. In *2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 724–733, 2023.

[32] J.A. van Niekerk, J.D. le Roux, and I.K. Craig. Reinforcement learning based automatic tuning of pid controllers in multivariable grinding mill circuits. *Control Engineering Practice*, 165:106522, 2025.

[33] B. van Werkhoven, J. Maassen, H.E. Bal, and F.J. Seinstra. Optimizing convolution operations on gpus using adaptive tiling. *Future Generation Computer Systems*, 30(1):14–26, 2014.

[34] Ben van Werkhoven. Kernel tuner: A search-optimizing gpu code auto-tuner. *Future Generation Computer Systems*, 90:347–358, 2019.

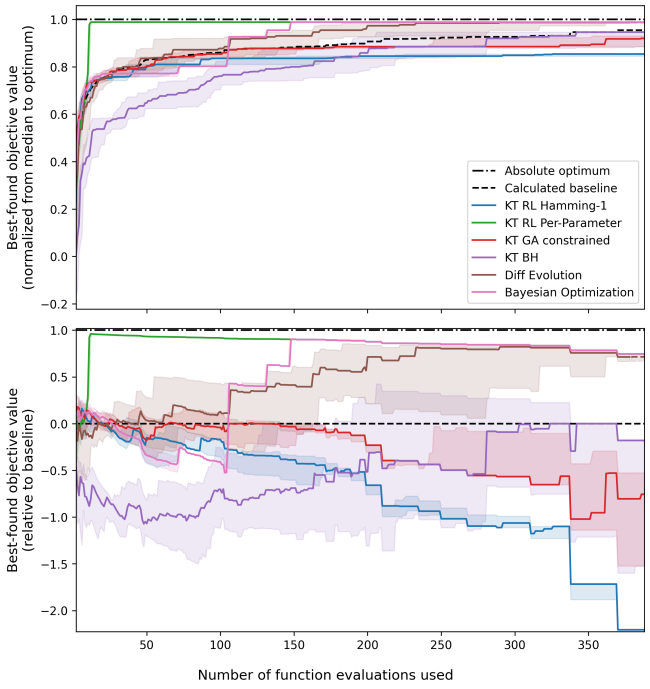
[35] David J. Wales and Jonathan P. K. Doye. Global optimization by basin-hopping and the lowest energy structures of lennard-jones clusters containing up to 110 atoms. *The Journal of Physical Chemistry A*, 101(28):5111–5116, 1997.

[36] Floris-Jan Willemsen, Richard Schoonhoven, Ji ří Filipovič, Jacob O. Tørring, Rob van Nieuwpoort, and Ben van Werkhoven. A methodology for comparing optimization algorithms for auto-tuning. *Future Gener. Comput. Syst.*, 159(C):489–504, October 2024.

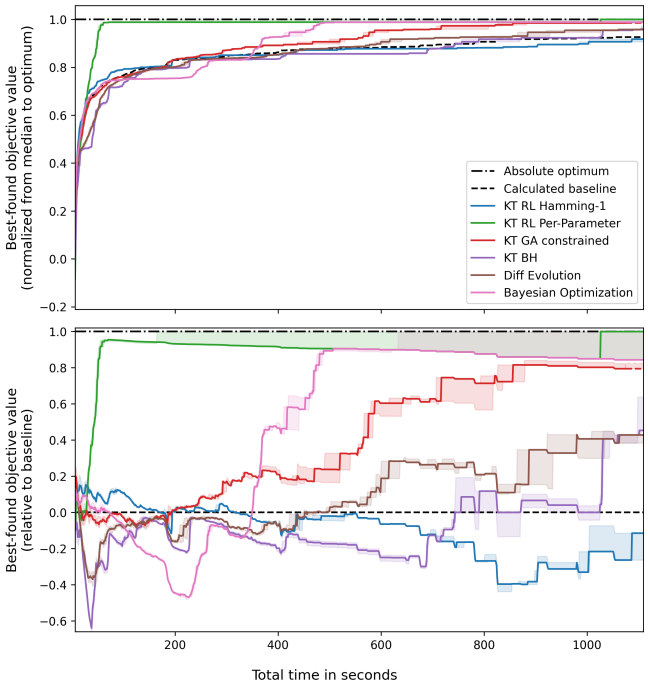
[37] Floris-Jan Willemsen, Rob van Nieuwpoort, and Ben van Werkhoven. Bayesian optimization for auto-tuning gpu kernels, 2021.

Appendices

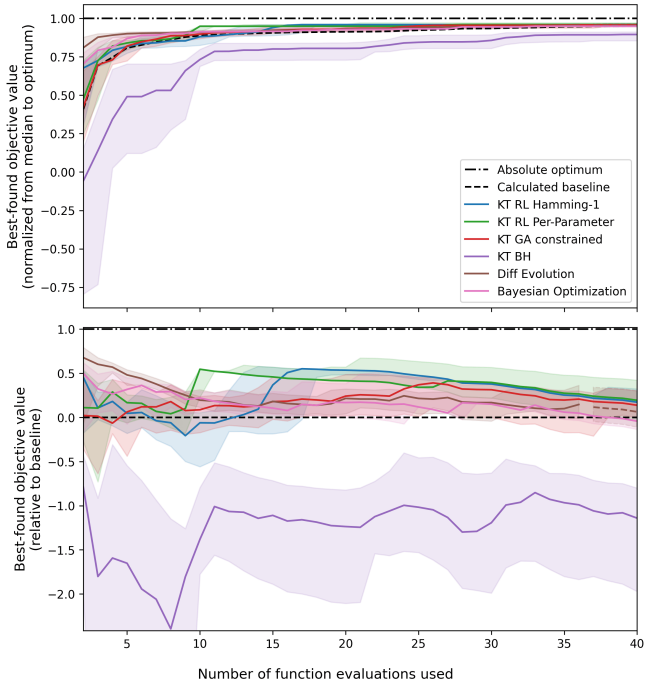
A Per-GPU Agents Results



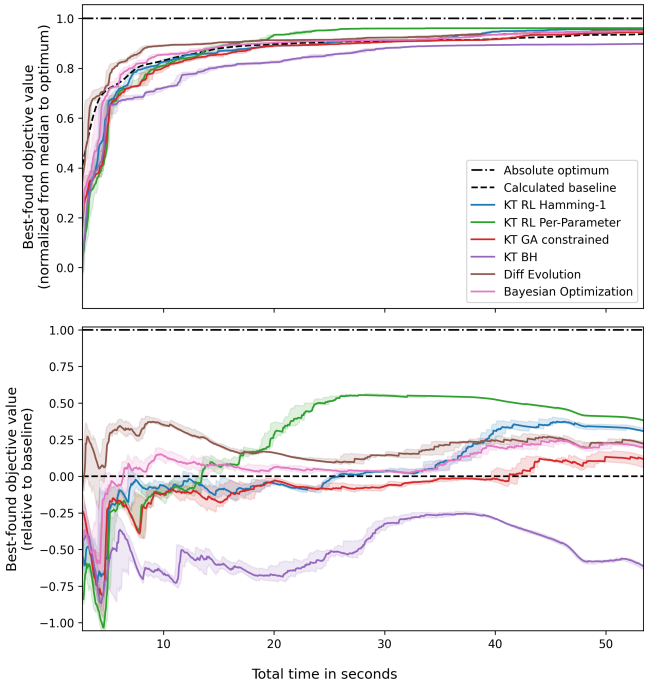
(a) Performance on A6000 over function evaluations



(b) Performance on A6000 over wall-clock time

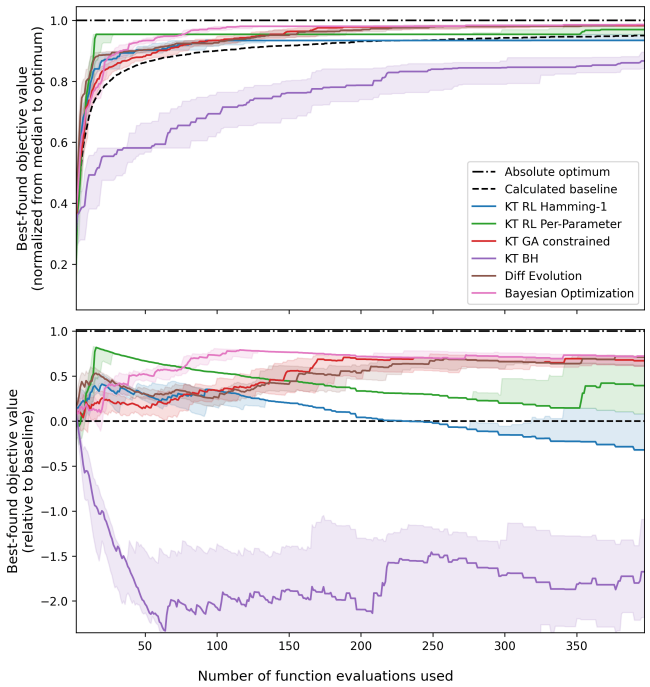


(c) Performance on W7800 over function evaluations

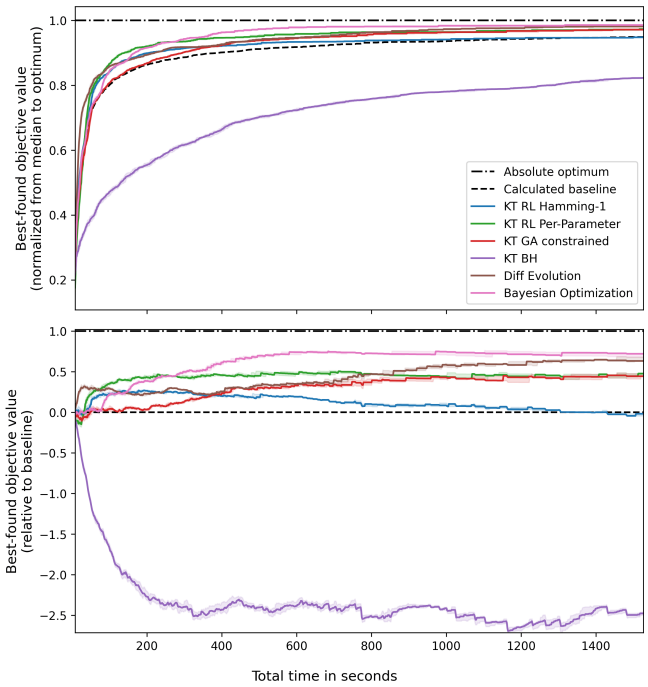


(d) Performance on W7800 over wall-clock time

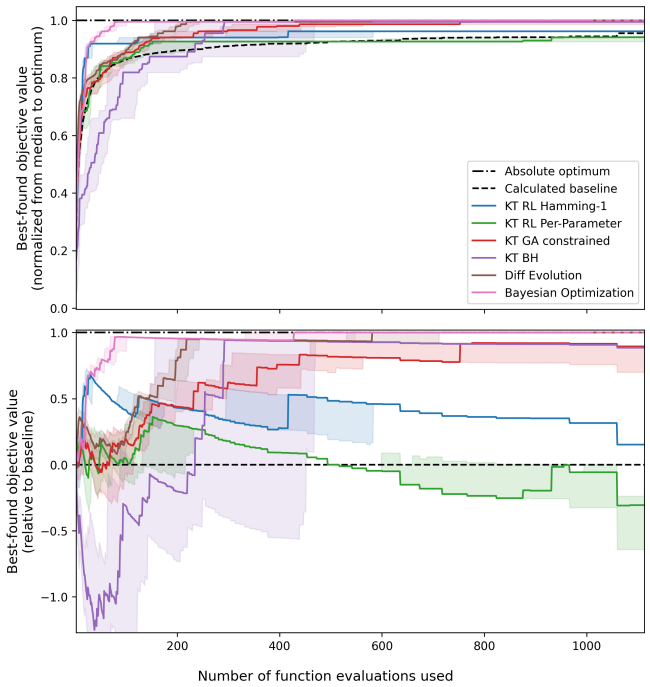
Figure 8: Results for Convolution kernel



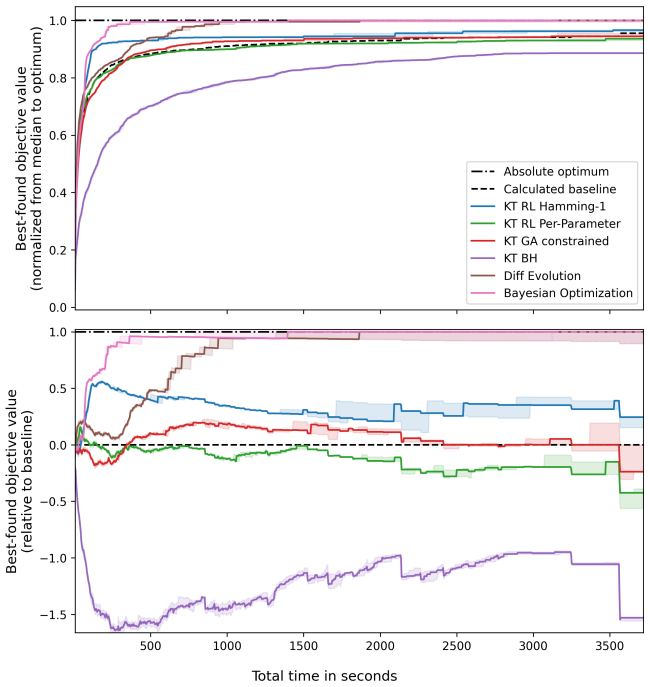
(a) Performance on A6000 over function evaluations



(b) Performance on A6000 over wall-clock time

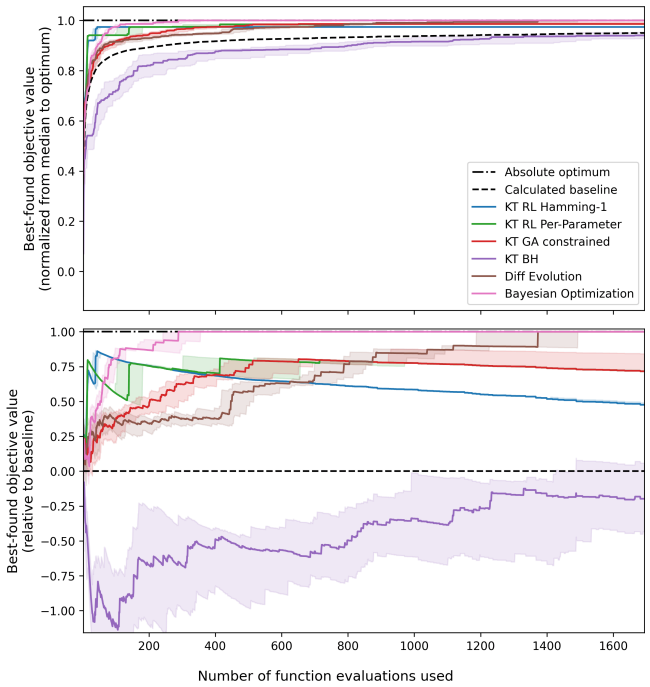


(c) Performance on W7800 over function evaluations

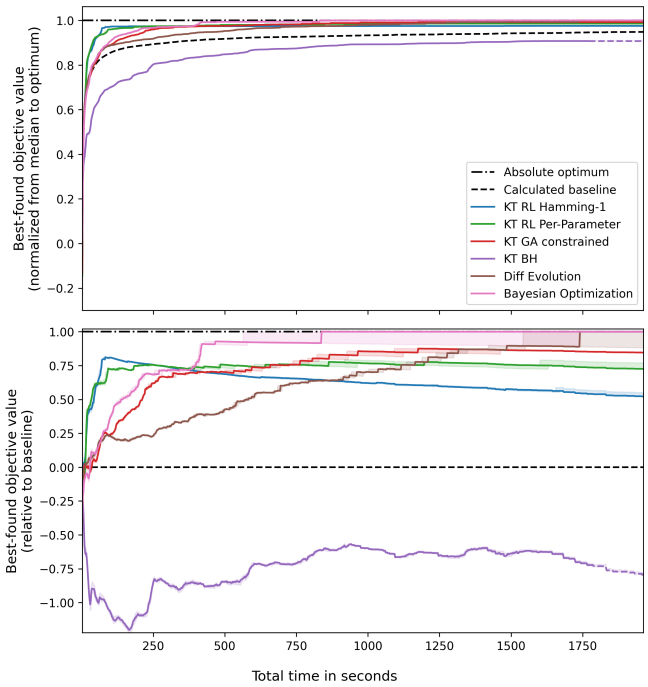


(d) Performance on W7800 over wall-clock time

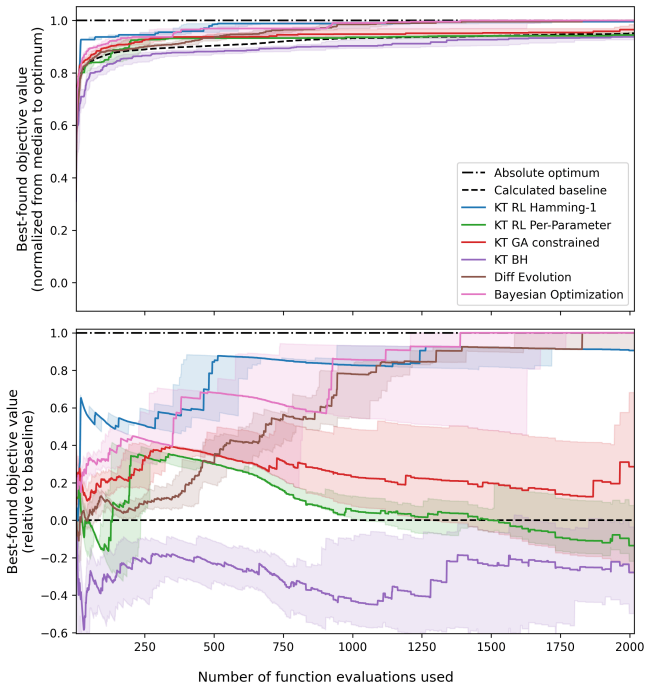
Figure 9: Results for dedispersion kernel



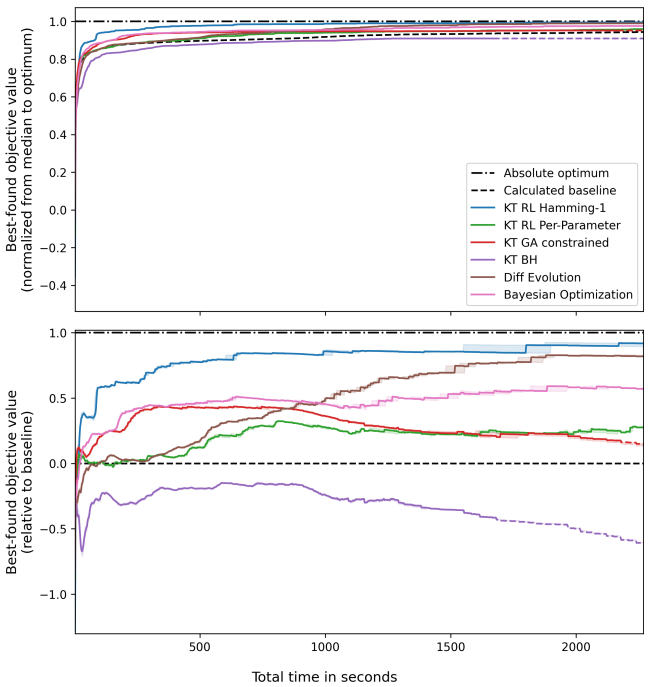
(a) Performance on A6000 over function evaluations



(b) Performance on A6000 over wall-clock time

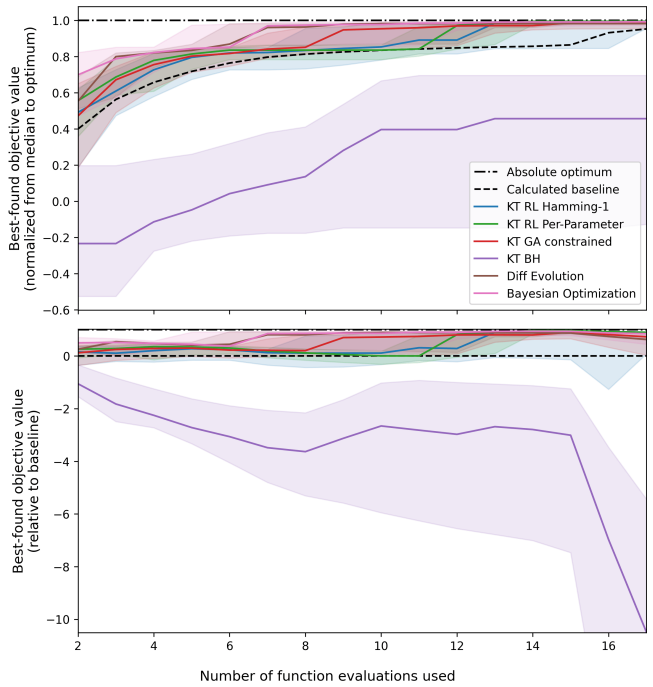


(c) Performance on W7800 over function evaluations

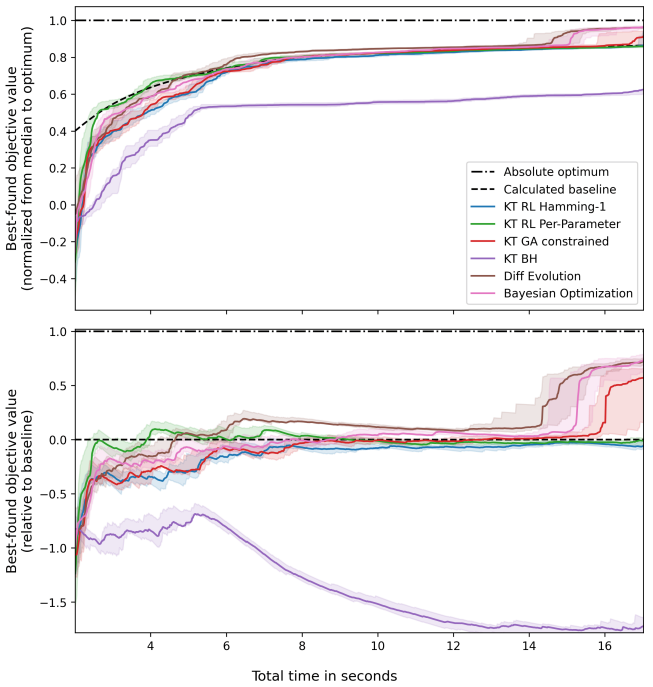


(d) Performance on W7800 over wall-clock time

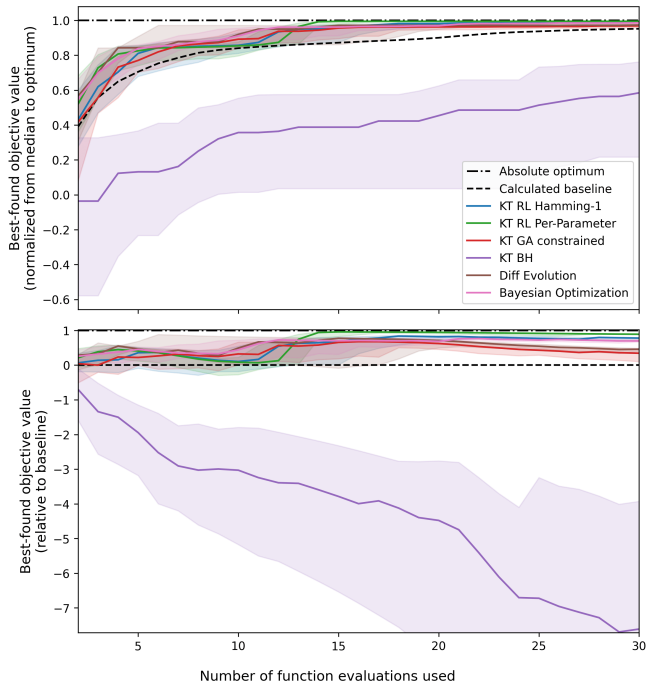
Figure 10: Results for gemm kernel



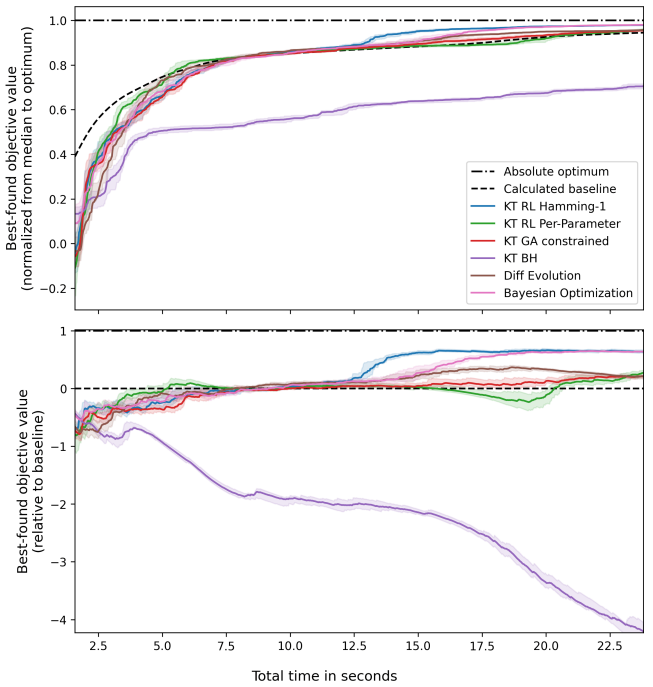
(a) Performance on A6000 over function evaluations



(b) Performance on A6000 over wall-clock time



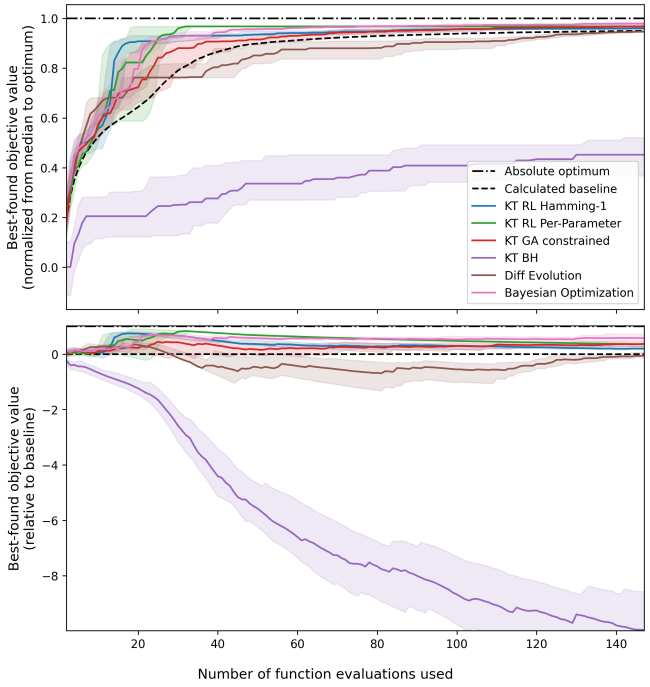
(c) Performance on W7800 over function evaluations



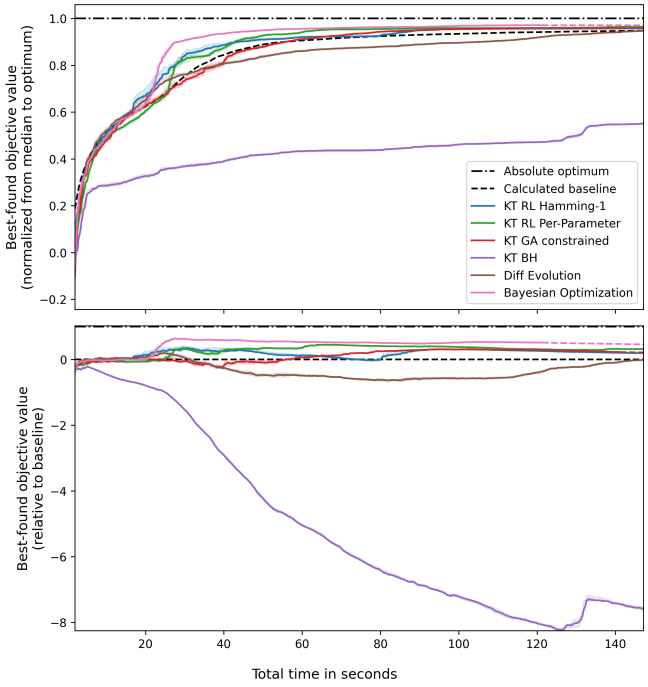
(d) Performance on W7800 over wall-clock time

Figure 11: Results for hotspot kernel

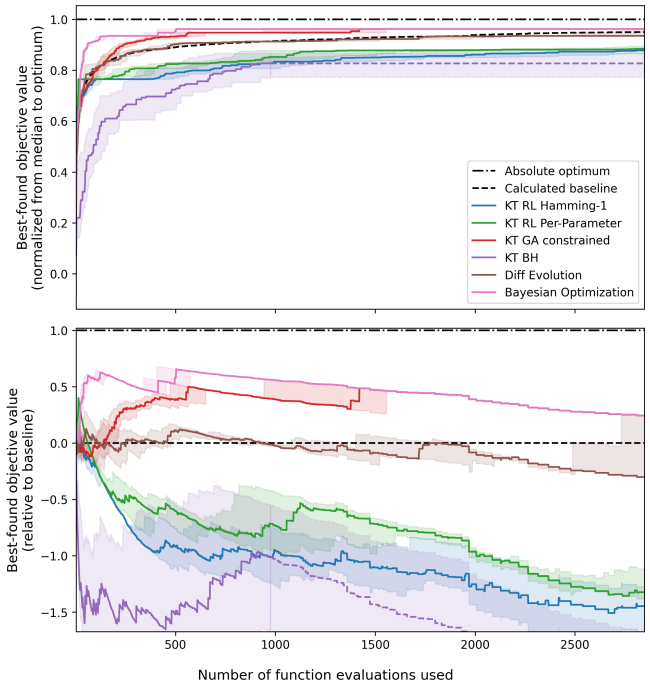
A.1 Hotspot Evaluation Results using GFLOPS/s as Objective



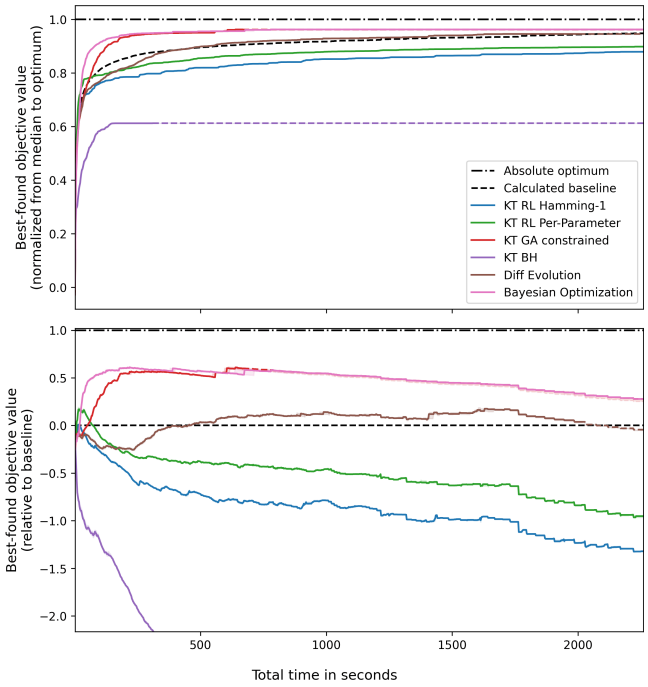
(a) Performance on A6000 over function evaluations



(b) Performance on A6000 over wall-clock time



(c) Performance on W7800 over function evaluations



(d) Performance on W7800 over wall-clock time

Figure 12: Results for hotspot kernel

B Model Train-Set Performance

The tables below show the selection metrics used to choose the model used for each kernel. This selection metric is the average gap between the median and optimal execution time, where 0.00% means the agent consistently found the optimum on average, and 100.00% means the agent finds the median as best configuration on average.

B.1 Hamming-1 Agent Train-Set Performance

Table 4: GPU performance results for hamming PPO seeds on Convolution

Model	A100	A4000	MI250X	W6600	GPU avg
hamming_1_ppo_seed0	21.01%	0.00%	0.00%	0.00%	5.25%
hamming_1_ppo_seed1	21.03%	0.00%	0.00%	0.00%	5.26%
hamming_1_ppo_seed2	0.00%	21.54%	6.28%	1.89%	7.43%
hamming_1_ppo_seed3	0.00%	20.08%	0.00%	0.00%	5.02%
hamming_1_ppo_seed4	0.00%	16.34%	0.00%	0.03%	4.09%
hamming_1_ppo_seed5	0.00%	0.70%	0.00%	0.00%	0.18%
hamming_1_ppo_seed6	20.33%	0.00%	0.00%	0.00%	5.08%
hamming_1_ppo_seed7	30.53%	0.00%	0.00%	0.00%	7.63%
GPU average over all models					4.99%

Table 5: GPU performance results for hamming PPO seeds on Dedispersion

Model	A100	A4000	MI250X	W6600	GPU avg
hamming_1_ppo_seed0	15.20%	20.51%	4.11%	5.47%	11.32%
hamming_1_ppo_seed1	0.00%	0.00%	0.00%	0.64%	0.16%
hamming_1_ppo_seed2	6.56%	0.66%	0.00%	0.00%	1.80%
hamming_1_ppo_seed3	0.96%	1.09%	0.00%	0.00%	0.51%
hamming_1_ppo_seed4	0.34%	0.00%	0.00%	1.56%	0.47%
hamming_1_ppo_seed5	40.91%	26.60%	50.42%	45.10%	40.76%
hamming_1_ppo_seed6	0.63%	0.00%	0.00%	0.00%	0.16%
hamming_1_ppo_seed7	29.77%	24.97%	3.59%	32.77%	22.77%
GPU average over all models					9.74%

Table 6: GPU performance results for hamming PPO seeds on GEMM

Model	A100	A4000	MI250X	W6600	GPU avg
hamming_1_ppo_seed0	0.00%	2.41%	0.00%	0.00%	0.60%
hamming_1_ppo_seed1	0.33%	0.00%	0.00%	19.85%	5.04%
hamming_1_ppo_seed2	0.33%	0.00%	1.19%	0.00%	0.38%
hamming_1_ppo_seed3	0.00%	0.00%	0.00%	0.00%	0.00%
hamming_1_ppo_seed4	0.09%	0.00%	0.00%	0.06%	0.04%
hamming_1_ppo_seed5	8.75%	6.21%	0.00%	0.00%	3.74%
hamming_1_ppo_seed6	0.00%	0.00%	0.00%	0.00%	0.00%
hamming_1_ppo_seed7	8.48%	4.79%	0.00%	0.00%	3.32%
GPU average over all models					1.64%

Table 7: GPU performance results for hamming PPO seeds on Hotspot

Model	A100	A4000	MI250X	W6600	GPU avg
hamming_1_ppo_seed0	22.41%	32.15%	25.83%	32.63%	28.26%
hamming_1_ppo_seed1	40.57%	44.91%	29.81%	40.11%	38.85%
hamming_1_ppo_seed2	41.37%	33.88%	39.74%	63.95%	44.73%
hamming_1_ppo_seed3	29.89%	35.11%	43.10%	37.25%	36.34%
hamming_1_ppo_seed4	72.93%	24.51%	49.63%	37.24%	46.08%
hamming_1_ppo_seed5	14.49%	39.48%	29.72%	39.78%	30.87%
hamming_1_ppo_seed6	44.13%	40.35%	21.50%	44.14%	37.53%
hamming_1_ppo_seed7	37.50%	30.42%	62.80%	53.06%	45.94%
GPU average over all models					38.58%

B.2 Per-parameter Agent Train-Set Performance

Table 8: GPU performance results for per-parameter PPO seeds on Convolution

Model	A100	A4000	MI250X	W6600	GPU avg
perparameter_ppo_seed0	0.00%	0.00%	0.00%	0.00%	0.00%
perparameter_ppo_seed1	0.00%	0.00%	0.00%	0.00%	0.00%
perparameter_ppo_seed2	0.00%	0.00%	0.00%	0.00%	0.00%
perparameter_ppo_seed3	0.00%	0.00%	0.00%	0.00%	0.00%
perparameter_ppo_seed4	0.00%	0.00%	0.00%	0.00%	0.00%
perparameter_ppo_seed5	2.10%	2.09%	0.00%	0.00%	1.05%
perparameter_ppo_seed6	0.00%	0.00%	0.00%	0.00%	0.00%
perparameter_ppo_seed7	0.00%	0.00%	0.00%	0.00%	0.00%
GPU average over all models					0.13%

Table 9: GPU performance results for per-parameter PPO seeds on Dedispersion

Model	A100	A4000	MI250X	W6600	GPU avg
perparameter_ppo_seed0	0.00%	0.00%	0.00%	0.00%	0.00%
perparameter_ppo_seed1	0.00%	0.00%	0.00%	0.00%	0.00%
perparameter_ppo_seed2	0.00%	0.00%	0.00%	0.00%	0.00%
perparameter_ppo_seed3	0.58%	0.00%	0.00%	0.00%	0.15%
perparameter_ppo_seed4	0.00%	0.00%	0.00%	2.32%	0.58%
perparameter_ppo_seed5	3.52%	0.00%	0.00%	0.00%	0.88%
perparameter_ppo_seed6	23.72%	0.11%	6.98%	0.00%	7.70%
perparameter_ppo_seed7	0.49%	5.98%	0.00%	1.29%	1.94%
GPU average over all models					1.41%

Table 10: GPU performance results for per-parameter PPO seeds on GEMM

Model	A100	A4000	MI250X	W6600	GPU avg
perparameter_ppo_seed0	0.00%	2.18%	0.00%	25.15%	6.83%
perparameter_ppo_seed1	0.00%	0.00%	10.87%	61.56%	18.11%
perparameter_ppo_seed2	8.29%	6.13%	0.00%	35.20%	12.41%
perparameter_ppo_seed3	0.00%	0.00%	10.95%	0.16%	2.78%
perparameter_ppo_seed4	21.53%	30.76%	241.07%	130.75%	106.03%
perparameter_ppo_seed5	0.00%	2.93%	3.38%	0.13%	1.61%
perparameter_ppo_seed6	26.23%	36.94%	16.77%	1.18%	20.28%
perparameter_ppo_seed7	5.95%	0.00%	0.00%	0.42%	1.59%
GPU average over all models					21.21%

Table 11: GPU performance results for per-parameter PPO seeds on Hotspot

Model	A100	A4000	MI250X	W6600	GPU avg
perparameter_ppo_seed0	0.00%	4.64%	2.01%	0.00%	1.66%
perparameter_ppo_seed1	0.00%	0.00%	1.91%	2.12%	1.01%
perparameter_ppo_seed2	0.00%	0.52%	7.70%	0.00%	2.06%
perparameter_ppo_seed3	0.00%	0.22%	0.00%	0.00%	0.06%
perparameter_ppo_seed4	3.46%	0.00%	0.74%	0.00%	1.05%
perparameter_ppo_seed5	0.00%	0.11%	0.00%	0.00%	0.03%
perparameter_ppo_seed6	3.12%	0.27%	0.74%	0.00%	1.03%
perparameter_ppo_seed7	58.03%	68.83%	54.94%	77.86%	64.92%
GPU average over all models					8.98%