Efficient Hyperparameter Tuning for Auto-tuning

Floris-Jan Willemsen[1,2] [0000−0003−2295−8263],
Rob V. van Nieuwpoort[1] [0000−0002−2947−9444], and
Ben van Werkhoven[1,2] [0000−0002−7508−3272]

[1] Leiden University, the Netherlands
[2] Netherlands eScience Center, the Netherlands

Abstract. Graphics Processing Units (GPUs) have become indispensable as a computing resource due to their exceptional computational performance. Programming for GPUs involves a vast number of choices, such as thread configurations, memory access patterns, and algorithmic variations, all of which can significantly impact performance. Auto-tuning is used to optimize the performance, accuracy, and energy efficiency of GPU programs by finding the best program variant among the many choices. With many possible combinations, efficient optimization algorithms are crucial for navigating these complex search spaces. The performance of these optimization algorithms is highly sensitive to their hyperparameters, necessitating a systematic approach to optimize them.
We present a novel method for general hyperparameter tuning of optimization algorithms for auto-tuning. In particular, we propose a robust statistical method for evaluating hyperparameter performance across search spaces, democratize hyperparameter tuning by providing the required software and data to the community, and present two orders of magnitude improvement over traditional hyperparameter tuning methods to make hyperparameter tuning scalable and feasible. Our hyperparameter tuning method substantially improves the performance of optimization algorithms in finding near-optimal configurations for GPU auto-tuning search spaces, even on problems not trained on, enabling successful auto-tuning on previously unattainable problem scales.

Keywords: Auto-tuning · Optimization · Hyperparameter Tuning

1 Introduction

Graphics Processing Units (GPUs) have revolutionized the computing landscape in the past decade, providing previously unattainable computational performance for compute-intensive tasks [7]. GPU programming models, such as HIP, OpenCL, and CUDA, allow developers to create highly parallel functions, called *kernels*, that run on the GPU. Developers are confronted with a myriad of implementation choices and optimization techniques related to thread organization, memory usage, and computation strategies to achieve optimal compute performance [9]. Many different design choices have a substantial and hard-to-predict impact on the performance of GPU kernels, as the optimal kernel configuration

depends on a complex interplay of hardware, device software, and the program itself. This optimization problem leads to an overwhelming number of code variants, too many to manually develop and explore, spurring the creation of frameworks that facilitate automatic performance tuning, or *auto-tuning*, to optimize GPU applications [1,12,6,13,15]. These auto-tuning frameworks rely on optimization algorithms to efficiently navigate the vast space of possible implementations, as exhaustivily attempting all combinations is costly.

The performance of these optimization algorithms in large part depends on so-called hyperparameters, the parameters of the optimization algorithms themselves. These hyperparameters can be tuned as well, iteratively evaluating configurations to find the best hyperparameter settings.

Despite the potential benefits of hyperparameter tuning optimization algorithms for auto-tuning problems, to the best of our knowledge, hyperparameter tuning is currently not supported by auto-tuning frameworks. This omission likely stems from fundamental differences between hyperparameter tuning in auto-tuning and general hyperparameter tuning when it comes to hardware considerations, the nature of the objective, and evaluation cost and scale.

First, unlike traditional hyperparameter tuning which is largely hardware-agnostic, hardware plays a crucial role in GPU auto-tuning, and optimization algorithms must be optimized across various GPU architectures to achieve good generalization, adding an additional dimension to the hyperparameter tuning.

Second, while traditional hyperparameter tuning generally measures time in abstract function evaluations, GPU auto-tuning must account for actual execution time, as it is directly impacted by the tuning itself. The resulting complex performance data must be aggregated across iterations and problems, and translated to a single measure of general performance.

Finally, evaluation cost and scale also differ significantly. Whereas model training can take hours or weeks, GPU auto-tuning involves compiling and executing kernels in seconds, typically reaching near-optimal solutions within minutes. However, the distribution and structure of the search spaces trained on can have a strong influence on optimization algorithm performance, the stochasticity of the optimization algorithms necessitates a high number of repeated executions, and achieving satisfactory general performance requires tuning on a wide array of search spaces, leading to stability and scalability challenges that are less prevalent in traditional hyperparameter tuning.

These differences necessitate an approach to hyperparameter tuning tailored to GPU auto-tuning frameworks. In this work, we introduce a novel method for efficient general hyperparameter tuning for GPU auto-tuning. We propose a systematic and general approach to hyperparameter optimization by expanding on our statistically robust methodology for comparing optimization algorithm performance for auto-tuning [17]. To make hyperparameter tuning feasible, we introduce a novel *simulation mode* for simulating optimization algorithms for auto-tuning, providing a two orders of magnitude improvement. Moreover, we provide and encourage reusable, reproducible, and shareable software and data for hyperparameter tuning based on emerging community standards. This de-

Table 1: Overview of auto-tuning frameworks, whether they are open source and actively maintained, supported optimization methods and support for setting hyperparameters without modifying the tuner source code.

Framework	Open Source	Active	Optimization algorithms	Hyperparameters
AUMA [5]	✓	✗	K-Bagging Neural Networks	✓(File-based)
CLTune [12]	✓	✗	Simulated Annealing, Particle Swarm Optimization, Neural Network	✓(API-based)
OpenTuner [1]	✓	✗	Simulated Annealing, Particle Swarm Optimization, Multi-armed Bandit, various Evolutionary/Genetic Algorithms, Local Search methods	✗
KTT [6]	✓	✓	Markov Chain Monte Carlo, Profile-based search	✗
ATF [13]	✓	✓	Simulated Annealing, Differential Evolution, Multi-armed Bandit, Local Search	✗
BaCO [8]	✓	✗	Bayesian Optimization, Exhaustive	✓(File-based)
Kernel Tuner [15]	✓	✓	20+ different global and local optimization algorithms, including Annealing methods, Genetic/Evolutionary methods, Swarm-based methods, and Bayesian Optimization	✓(API-based)

mocratizes hyperparameter tuning, enables rapid experimentation, and reduces cost and energy consumption. Our contributions have been implemented in Kernel Tuner [15,16], an open-source framework for auto-tuning GPU applications.

The remainder of this work is structured as follows. Section 2 discusses related work. Section 3 describes the context, design, and implementation of our approach to hyperparameter tuning for auto-tuning. In Section 4, we evaluate the impact, generalization, and scalability of our method using a wide variety of real-world search spaces. Section 5 concludes this work.

2 Related Work

In this section, we discuss related works to provide context on the developments in and current state of auto-tuning hyperparameter tuning. There are many different automated approaches to improving the performance of software that are collectively referred to as auto-tuning. For a survey of different uses of auto-tuning in high-performance computing, see Balaprakash et al. [4].

We will focus on various auto-tuning frameworks and features regarding optimization algorithms and hyperparameters. Table 1 outlines various contemporary auto-tuning frameworks and relevant characteristics. It is interesting that most frameworks listed do not have a convenient way to set hyperparameters, if at all possible. Ashouri et al. [2] wrote a comprehensive survey on machine-learning methods for auto-tuning, yet this does not mention hyperparameter tuning. It consequently appears that the impact of hyperparameter tuning in auto-tuning is largely understudied.

The most closely related work is perhaps [14], in which a limited hyperparameter tuning is conducted to compare optimization algorithms for GPU auto-tuning. They measure performance by counting head-to-head wins between two algorithms on individual problems and select hyperparameter tuning configura-

tions by repeatedly combining configurations with the most wins from individual problems. The impact of the hyperparameter tuning is not evaluated.

3 Design and Implementation

This section discusses the context (Section 3.1), design (Section 3.2), feasibility (Section 3.3), and implementation details (Section 3.4) of our novel method for efficient hyperparameter tuning for GPU auto-tuning.

3.1 Context

Auto-tuning involves optimizing a kernel K_i on a GPU G_j for an input dataset I_k to maximize performance $f_{G_j,I_k}(K_i)$. The auto-tuner constructs a search space $\mathcal{X}$ by considering all tunable parameters and their valid values, subject to user-defined constraints. The objective is to determine the optimal configuration, or more formally (assuming minimization) as follows:

$$x^\star = \arg\min_{x \in \mathcal{X}} f_{G_j,I_k}(K_{i,x}). \tag{1}$$

The evaluation of each configuration takes time, as it needs to be compiled and executed on the GPU. As search spaces in GPU auto-tuning tend to contain a large number of configurations, it is generally not feasible to evaluate all configurations. In addition, the GPU auto-tuning search spaces are generally discontinuous, non-convex, and highly irregular, requiring optimization algorithms suitable for this context to navigate the search spaces. Poorly adapted search strategies cause excessive evaluations, making tuning infeasible for practical use.

3.2 Hyperparameter Tuning

To improve the performance of optimization algorithms in auto-tuning, we propose to employ hyperparameter tuning. Hyperparameter tuning adjusts the settings of an optimization algorithm to enhance its efficiency across a domain of problems. Examples of hyperparameters include population size in evolutionary algorithms, neighbor selection in local search methods, and temperature in annealing-based approaches. Unlike auto-tuning, which optimizes parameters within a single search space, hyperparameter tuning aims for *general optimal performance across multiple tuning problems.*

The methodology for evaluating optimization algorithms, introduced in [17], provides a systematic approach to comparing search strategies across search spaces. It defines a performance metric $\mathcal{P}$ which quantifies an optimization algorithm's performance over the passed time relative to a calculated baseline for comparison across search spaces. The hyperparameter tuning problem can be formalized as:

$$h^\star = \arg\max_{h \in H} \mathcal{P}(\mathcal{F}_h, K, G, I), \tag{2}$$

where H represents the hyperparameter space, and $\mathcal{F}_h$ is the optimization algorithm configured with hyperparameters h. The kernels K, GPUs G, and inputs I are the collections of K_i, G_j, and I_k of Eq. (1) on which the hyperparameter tuning is conducted (also known as the training set). The usage of $\mathcal{P}$ provides a statistically robust metric for hyperparameter tuning, enabling optimizing the optimization algorithm performance across search spaces.

3.3 Hyperparameter Tuning Feasibility

It must be noted that evaluating Eq. (2) requires that for each hyperparameter configuration h in H, an auto-tuning experiment must be run on each combination of the involved kernels, GPUs, and input datasets, leading to a combinatorial explosion in computational cost. In addition, this implies that the various hardware configurations are available for the full duration of the hyperparameter tuning. Moreover, many optimization algorithms are stochastic, requiring repeated evaluations to ensure a reliable outcome of the hyperparameter tuning. The combination of these factors makes executing such a hyperparameter tuning prohibitively expensive and time-consuming.

To address this challenge, we propose a so-called simulation mode. Instead of running the recurring auto-tuning runs required for hyperparameter tuning directly on the hardware, this simulation mode retrieves pre-collected performance data from a cache file, mimicking the behavior of real auto-tuning experiments. For such a simulation mode to work, all configurations in a search space must have been evaluated on the actual hardware (a brute-force auto-tuning search). This might seem counter-intuitive as we are optimizing algorithms to prevent such brute-force searches when auto-tuning in the first place. To illustrate why this is sensible in this case, imagine hyperparameter tuning a stochastic algorithm with 100 possible hyperparameter configurations for 1000 function evaluations, each run of the algorithm repeated 25 times due to stochasticity, on 3 kernels across 3 GPUs. In addition, assume each of the 9 resulting search spaces have one million valid configurations, that each take one second to compile and execute. Brute-forcing each search space would take $\sim$ 11.6 days, or $\sim$ 104 compute days for all 9 search spaces, although the individual search spaces can be brute forced in parallel. In contrast, hyperparameter tuning a single optimization algorithm in the same scenario with live auto-tuning runs would take 260 compute days. And that is just for a single optimization algorithm; as seen in Table 1, most auto-tuning frameworks provide multiple optimization algorithms.

The simulation mode approach provides several important advantages. First, it substantially improves scalability: after the high one-off cost of brute-forcing, the threshold of comparison to and hyperparameter tuning of additional optimization algorithms is substantially lower. In addition, it mitigates measurement noise: the simulation mode ensures that every auto-tuning execution is reproducible, preventing additional noise in the process. Finally, it reduces computational and energy costs and limitations: once the full search space is evaluated, hyperparameter tuning can be performed without the need for costly compila-

tion and execution on the GPU, and without occupying those resources during the hyperparameter tuning.

To ensure the simulation mode is representative of real-world auto-tuning performance, we collect execution data across a diverse set of tuning problems and hardware. This dataset serves as a foundation for evaluating hyperparameter tuning strategies without the need for real-time benchmarking, enabling rapid experimentation and algorithm comparison and reducing cost and energy consumption. In addition, making this publically available provides and promotes a reproducible record that can be used and contributed to by the community.

3.4 Integration with Kernel Tuner

To demonstrate the real-world impact of our proposed method, we implement it in the open-source GPU auto-tuning framework Kernel Tuner. Kernel Tuner is a Python-based auto-tuner designed to optimize GPU kernels for various objectives, such as execution time and energy efficiency [15]. It supports CUDA, HIP [11], OpenCL, and OpenACC for both C and Fortran, making it a versatile tool for GPU developers. With over 20 optimization algorithms, Kernel Tuner provides flexibility in searching large optimization spaces efficiently [14]. The wide variety of implemented optimization algorithms provides a large potential gain in performance by hyperparameter tuning, making Kernel Tuner an appropriate candidate for implementing our hyperparameter tuning method.

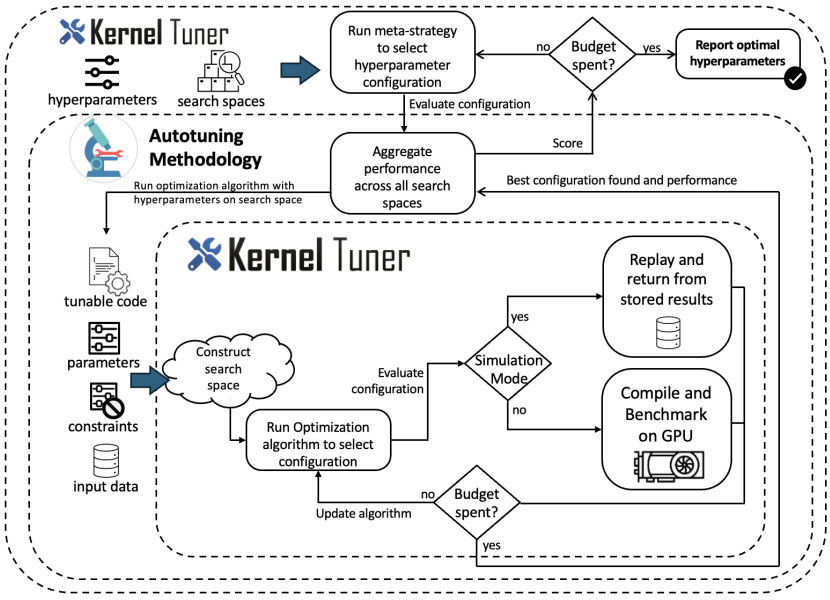


Fig. 1: The hyperparameter tuning pipeline for auto-tuning. Kernel Tuner's hyperparameter tuning functionality (the outermost layer) calls the autotuning methodology software to get an aggregate performance score, which is obtained by running the optimization algorithm on various search spaces.

An abstract visualization of our hyperparameter tuning pipeline is shown in Fig. 1. The hyperparameter tuning we implemented in Kernel Tuner uses the autotuning methodology [17] software package to request a performance score for an optimization algorithm with a selected hyperparameter configuration across the selected search spaces. The autotuning methodology package in turn uses our new simulation mode to run the optimization algorithm with the hyperparameter configuration on each of the search spaces. This process is repeated until the set tuning budget is spent or all hyperparameter configurations have been evaluated.

We implemented the simulation mode by extending Kernel Tuner's existing checkpointing mechanism. Each segment in the process of evaluating an auto-tuning configuration is registered, such as the time spent by the optimization algorithm, compilation, execution, and framework overhead, providing a trace of an auto-tuning run that can be replayed. For each auto-tuning configuration evaluated, Fig. 1 illustrates how the simulation mode integrates with Kernel Tuner's architecture. A dedicated simulation runner serves cached performance data, ensuring that the simulated execution accurately reflects real-world tuning times. Additionally, we modify the stopping criteria of optimization algorithms to account for simulated time, ensuring fair comparisons across different tuning strategies. During simulation mode, the optimization algorithm selects a configuration, upon which the trace recorded earlier is replayed to get the result and update the tuning budget as if it had been executed. From the point of view of the optimization algorithm, there is no perceivable difference between live tuning and the simulation mode. The simulation mode is integrated into Kernel Tuner's workflow, benefitting users by allowing switching between live and simulated tuning runs. By enabling efficient and repeatable evaluation of optimization algorithms, the simulation mode makes large-scale hyperparameter tuning feasible without excessive resource consumption.

To efficiently explore hyperparameter configurations, we also extended Kernel Tuner to support using its optimization algorithms as meta-strategies for hyperparameter tuning. This enables structured searches for hyperparameter tuning without reliance on brute-force methods, allowing near-optimal hyperparameter configurations to be found more efficiently.

Our implementation is compatible with data standards formed by the auto-tuning community [10], enabling anyone to easily apply our method for scalable general hyperparameter tuning.

4 Evaluation

In this section, we evaluate the advancements presented in Section 3 to determine the efficacy and scalability of our hyperparameter tuning method. Section 4.1 details the experimental setup. In Section 4.2, the results are discussed to draw conclusions on the performance of our method; first on the impact of hyperparameter tuning with the training set, followed by the generalization using the test set, and finally on the scalability of the simulation mode compared to live hyperparameter tuning.

4.1 Experimental Setup

To obtain a diverse set of real-world cases to evaluate our method on, we use four real-world auto-tuning applications on five different GPUs, resulting in 20 unique search spaces. The four applications are the dedispersion, convolution, hotspot, and GEMM kernels as described by [11]. The five GPUs are an AMD MI250X (in the LUMI supercomputer), AMD W6600, Nvidia A6000, Nvidia A4000, and Nvidia A100 (the latter four in the DAS-6 supercomputer). Each application is fully brute-force auto-tuned on each GPU system, after which all other evaluations are performed on the sixth-generation DAS supercomputer [3] using a Nvidia A4000 GPU node, demonstrating that once an exhaustive auto-tuning is conducted, access to the original systems is no longer needed, an important advantage of our method. This node has a 24-core AMD EPYC-2 7402P CPU, 128 GB of memory, and is running Rocky Linux 4.18. For a fair evaluation, the hyperparameter tuning is conducted on a training set of twelve search spaces resulting from the four applications on the AMD MI250X, Nvidia A100, and Nvidia A4000, while the test set consists of the eight search spaces resulting from the four applications on the AMD W6600 and Nvidia A6000.

Table 2: Hyperparameters for the tuned optimization algorithms. The optimal values are emphasized.

Algorithm	**Hyperparameter**	**Values**
Differential Evolution	method	{best1bin, best1exp, rand1exp, randtobest1exp, best2exp, rand2exp, **randtobest1bin**, best2bin, rand2bin, rand1bin}
	popsize	{10, **20**, 30}
	maxiter	{50, **100**, 150}
Dual Annealing	method	{**COBYLA**, L-BFGS-B, SLSQP, CG, Powell, Nelder-Mead, BFGS, trust-constr}
Genetic Algorithm	method	{single_point, two_point, **uniform**, disruptive_uniform}
	popsize	{10, 20, **30**}
	maxiter	{**50**, 100, 150}
	mutation_chance	5, 10, **20**}
Particle Swarm Optimization (*PSO*)	popsize	{10, **20**, 30}
	maxiter	{50, 100, **150**}
	c1	{1.0, 2.0, **3.0**}
	c2	{0.5, 1.0, **1.5**}

While Kernel Tuner provides a plethora of optimization algorithms as seen in Table 1, due to space constraints we select the four optimization algorithms shown in Table 2 to represent the diverse range of global optimization algorithms and hyperparameters available. Differential evolution is a population-based evolutionary algorithm that generates new candidate solutions by applying vector-based mutations and recombinations. Dual annealing combines elements of simulated annealing with local search. Genetic algorithm evolves solutions over generations using selection, crossover, and mutation operators. Particle swarm optimization (*PSO*) navigates optimization landscapes through information sharing among candidate solutions. A sensitivity test of the hyperparameters using the non-parametric Kruskal-Wallis test and mutual information scoring revealed that

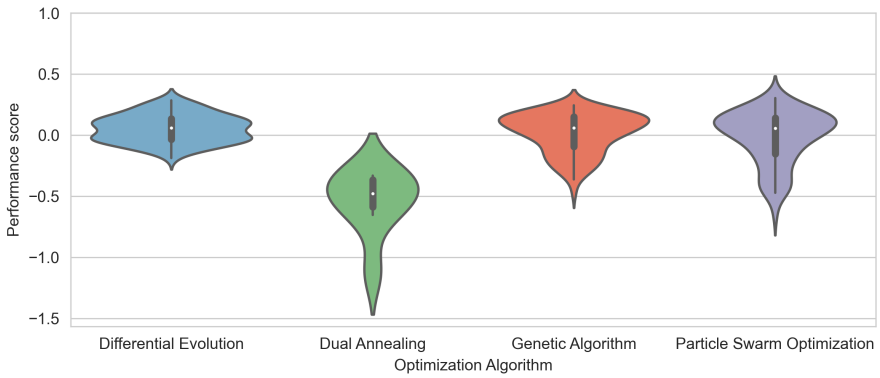


Fig. 2: Violin plots of the performance scores for all hyperparameter configurations of the evaluated optimization algorithms, showing the mean (white dot), boxplot (black box) and distribution (area).

the W hyperparameter of PSO has no meaningful effect on the score, and as such is left out. Each combination of hyperparameter values shown in Table 2 is run 25 times on each of the 12 search spaces to obtain a stable performance score. This means that tuning the hyperparameters of e.g. Genetic Algorithm requires running the algorithm 32400 times. The allocated budget for each run is equivalent to the time it takes the baseline to reach 95% of the distance between the search space median and optimum, per the methodology presented in [17]. In the performance comparisons of Section 4.2, each instance is re-executed with 50 repeats to mitigate stochasticity.

4.2 Results

First, we discuss the results regarding the impact of our hyperparameter tuning method. Figure 2 depicts the distribution of hyperparameter configuration performance scores for each optimization algorithm. With these scores, which will be used throughout this evaluation, an algorithm with a score of 0 performs as well as the calculated baseline, while a score of 1 indicates that the optimum is found immediately. When comparing two scores directly, a difference of 10% in the score can mean that given the same wallclock time budget, the algorithm with the higher score finds configurations that are 10% closer to the optimum. For more information on the performance scores, see [17]. The violin plots in Fig. 2 depict large differences in scores between the various hyperparameter configurations within the optimization algorithm. In addition, it shows the difference in sensitivity to hyperparameter tuning between optimization algorithms. An average improvement in performance score of 120.6% highlights the potential performance improvement gained by hyperparameter tuning.

Next, we examine the overall performance and generalization of the hyperparameter tuning across the search spaces. Figure 3 shows that the best performing configuration obtained in tuning remains largely stable when re-executed

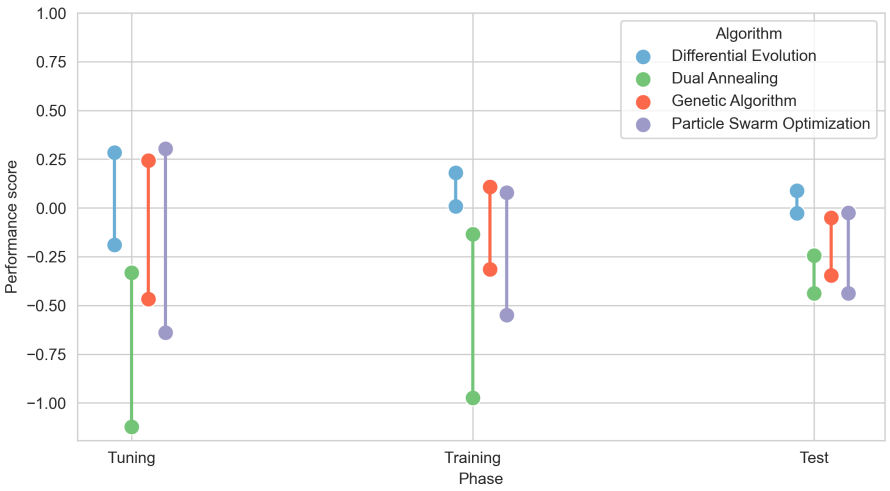
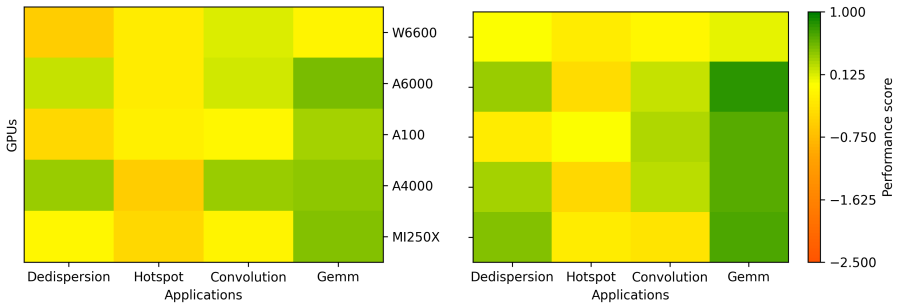


Fig. 3: Best and worst scores on tuning, training, and test for evaluated optimization algorithms.

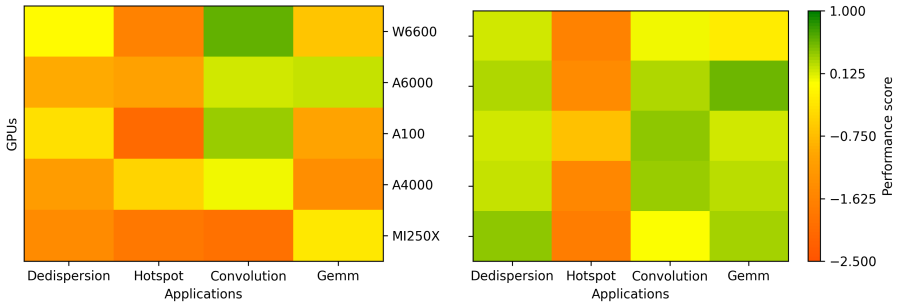
on a higher resolution on the same training set, while the configurations with worst performance perform relatively better, as stochastic effects are evened out. The best configuration performance is only slightly diminished on the test set, as should be expected, indicating good generalization performance. In Fig. 4, visual inspection of both the *untuned* (where the hyperparameters have been optimized for worst performance instead of best) and tuned for each of the optimization algorithms reveals that each of the tuned versions improves upon their counterpart in general, instead of over-optimizing on a limited number of search spaces to boost the score. Most importantly, it can be seen that performance is improved in both the search spaces trained on (those with *MI250X*, *A100* and *A4000* GPUs) as well as the search spaces that have not been tuned on (those with *W6600* and *A6000* GPUs).

Moreover, the aggregate general performance is shown in Fig. 5, where it can be seen that over time the tuned optimization algorithms outperform their untuned counterparts by a wide margin. Quantifying this on the test search spaces, Differential Evolution is improved by 0.116, Dual Annealing by 0.192, Genetic Algorithm by 0.295, and PSO by 0.411, for an average improvement of 81.3%. Quantified on all search spaces, Differential Evolution is improved by 0.151, Dual Annealing by 0.573, Genetic Algorithm by 0.353, and PSO by 0.53, for an average improvement of 101.9%, demonstrating that our hyperparameter tuning method results in substantial generalized performance improvement.

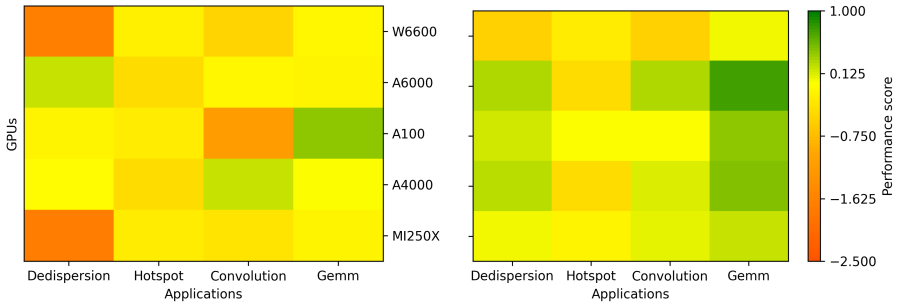
Finally, as to the scalability of our method, we can compare it to the estimated times without the simulation mode, as is shown in Fig. 6. With our method, hyperparameter tuning took approximately 4.5 hours for the optimization algorithm with the least hyperparameter configurations (*Dual Annealing*) and 54 hours for the optimization algorithm with the most hyperparameter con-



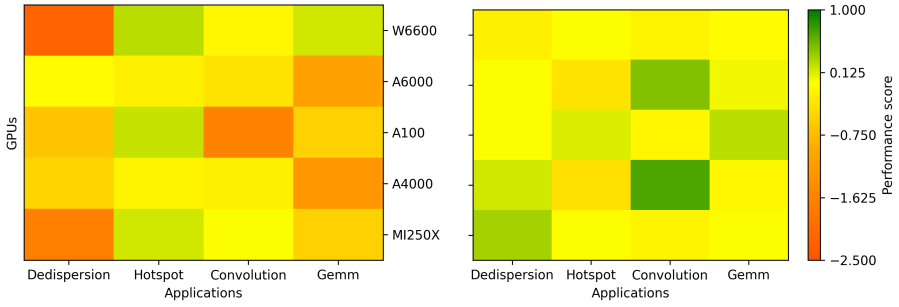
(a) Differential Evolution untuned (b) Differential Evolution tuned



(c) Dual Annealing untuned (d) Dual Annealing tuned



(e) Genetic Algorithm untuned (f) Genetic Algorithm tuned



(g) PSO untuned (h) PSO tuned

Fig. 4: Impact of tuning on optimization algorithm performance per search space.

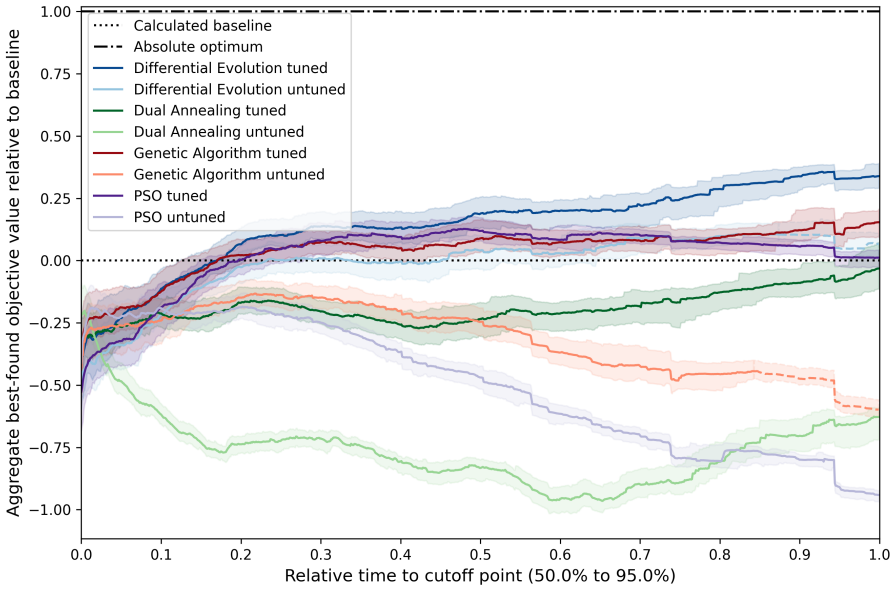


Fig. 5: Aggregate performance over time between untuned and tuned optimization algorithms accross all search spaces.

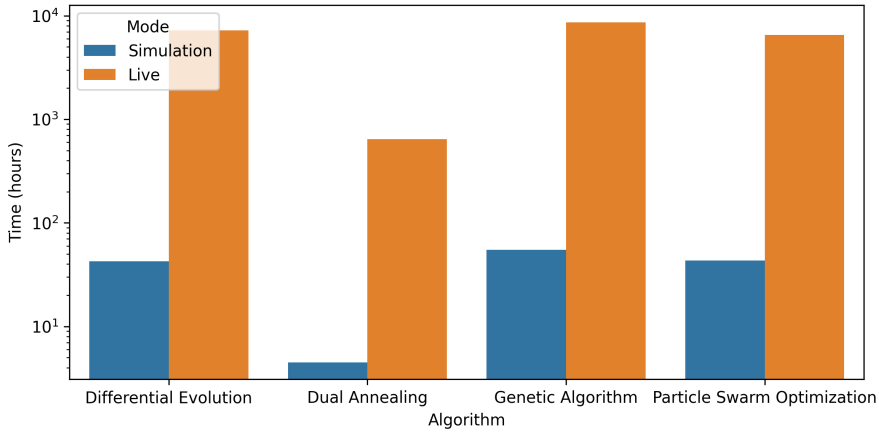


Fig. 6: Tuning time comparison between live and simulation mode per optimization algorithm.

figurations (*Genetic Algorithm*). While this requires the upfront cost of brute-forcing each search space, this is a one-off cost and allows the hyperparameter tuning of multiple optimization algorithms in parallel. In contrast, without the simulation mode, the hyperparameter tuning would take approximately 642 hours (nearly a month) for the optimization algorithm with the least hyperparameter configurations and 8672 hours (nearly a year) for the optimization algorithm with the most hyperparameter configurations. In total, live-tuning the four optimization algorithms of this evaluation would take 23045 hours (nearly three years), whereas it took 145 sequential hours using the simulation mode.

We thus conclude based on this evaluation that our method for hyperparameter tuning for auto-tuning optimization algorithms has a substantial effect on the performance, generalizes well beyond the training data, and is scalable.

5 Conclusion

In this work, we have introduced a novel approach to hyperparameter tuning for optimization algorithms used in GPU auto-tuning. As GPU architectures and applications continue to grow in complexity, efficient auto-tuning is crucial to fully leverage the computational power of modern GPUs, which in turn depends on optimization algorithm performance. Our method addresses the challenges of hyperparameter optimization in this domain by building upon a statistically robust methodology for comparing optimization algorithms, introducing a simulation mode to enable scalable hyperparameter tuning, and promoting the use of reproducible and shareable benchmark resources.

The results demonstrate that our approach substantially improves the efficiency of optimization algorithms in finding near-optimal configurations in large auto-tuning search spaces and that these improvements hold on search spaces not trained on. By systematically tuning the hyperparameters of these algorithms, we improved their overall performance by 101.9% on average. Our simulation mode reduces the computational cost of hyperparameter tuning by two orders of magnitude, making large-scale experiments feasible and preventing the need for constant access to hardware and excessive resource usage. By addressing the efficacy, scalability, and reproducibility challenges in hyperparameter tuning, this work contributes to the advancement of auto-tuning, enabling more efficient utilization of modern GPUs in scientific and industrial applications.

In conjunction with this work, we share our extensive data set[3] in accordance with community standards to be used by and contributed to by the community.

Kernel Tuner can be installed with `pip install kernel-tuner`. The optimized hyperparameters that have been found in this work have been set as defaults in Kernel Tuner to benefit its users. For more information or to contribute, please visit the GitHub repository[4].

Acknowledgments. The CORTEX project has received funding from the Dutch Research Council (NWO) in the framework of the NWA-ORC Call (file #NWA.1160.18.316).

[3] https://github.com/AutoTuningAssociation/benchmark_hub
[4] https://github.com/KernelTuner/kernel_tuner

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Ansel, J., et al: OpenTuner: An extensible framework for program autotuning. 2014 23rd Int. Conf. Parallel Archit. Compil. Tech. PACT (2014)
2. Ashouri, A.H., et al: A Survey on Compiler Autotuning using Machine Learning. ACM Comput. Surv. (2019)
3. Bal, H., et al: A medium-scale distributed system for computer science research: Infrastructure for the long term. Computer (2016)
4. Balaprakash, P., et al: Autotuning in High-Performance Computing Applications. Proc. IEEE (2018)
5. Falch, T.L., Elster, A.C.: Machine learning based auto-tuning for enhanced OpenCL performance portability. In: 2015 IEEE Int. Parallel Distrib. Process. Symp. Workshop (2015)
6. Filipovič, J., et al: Autotuning of OpenCL kernels with global optimizations. In: Proc. 1st Workshop Autotuning Adapt. Approaches Energy Effic. HPC Syst. (2017)
7. Heldens, S., et al: The Landscape of Exascale Research: A Data-Driven Literature Analysis. ACM Comput. Surv. (2021)
8. Hellsten, E.O., et al: BaCO: A fast and portable bayesian compiler optimization framework. In: Proc. 28th ACM Int. Conf. Archit. Support Program. Lang. Oper. Syst. Vol. 4 (2024)
9. Hijma, P., et al: Optimization Techniques for GPU Programming. ACM Comput. Surv. (2023)
10. Hozzová, J., et al: FAIR sharing of data in autotuning research (vision paper). In: Companion 15th ACMSPEC Int. Conf. Perform. Eng. (2024)
11. Lurati, M., et al: Bringing Auto-Tuning to HIP: Analysis of Tuning Impact and Difficulty on AMD and Nvidia GPUs. In: Euro-Par 2024 Parallel Process. (2024)
12. Nugteren, C., Codreanu, V.: CLTune: A generic auto-tuner for OpenCL kernels. 9th Int. Symp. Embed. MulticoreMany-Core Syst.–Chip (2015)
13. Rasch, A., Gorlatch, S.: ATF: A generic directive-based auto-tuning framework. Concurr. Comput. Pract. Exp. (2018)
14. Schoonhoven, R., et al: Benchmarking optimization algorithms for auto-tuning GPU kernels. IEEE Trans. Evol. Computat. (2022)
15. van Werkhoven, B.: Kernel Tuner: A search-optimizing GPU code auto-tuner. Future Generation Computer Systems (2019)
16. Willemsen, F.J., et al: Bayesian Optimization for auto-tuning GPU kernels. In: 2021 PMBS (2021)
17. Willemsen, F.J., et al: A methodology for comparing optimization algorithms for auto-tuning. Future Gener. Comput. Syst. (2024)