Efficient Hyperparameter Tuning for Auto-tuning

Floris-Jan Willemsen[1,2][0000−0003−2295−826],
Rob 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 for data- and compute-intensive tasks. 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 selecting the best program variant from the many choices. As GPU architectures and applications grow in complexity, auto-tuners face increasing demands. With millions of possible combinations, efficient optimization algorithms are crucial for navigating the search space. The performance of these optimization algorithms is exceedingly sensitive to their hyperparameters, necessitating a systematic approach to optimize them. In this work, we present a novel method for general hyperparameter tuning of optimization algorithms for auto-tuning. In particular, we apply a robust statistical method for comparing optimization algorithm performance across search spaces, democratize hyperparameter tuning by providing resources to the community, and introduce a simulation mode to make our hyperparameter tuning on a wide variety of large search spaces feasible. The efficacy of this method is demonstrated by performance comparisons on various real-world applications. Our hyperparameter tuning method for auto-tuning improves the efficiency of optimization algorithms in finding near-optimal configurations for GPU auto-tuning search spaces, enabling 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 such as artificial intelligence and climate simulation [7,12]. GPUs excel in terms of compute performance and energy efficiency for tasks that involve large data sets and dense computation, making them increasingly vital in various scientific domains [19]. In the past decade, GPUs have become increasingly complex computing devices with larger register files, more specialized cores, and larger and more complex streaming multiprocessors (SMs), while also dramatically increasing the number of SMs per chip [10]. In addition, energy efficiency and accuracy play an increasingly important role in the auto-tuning of GPU applications [16,8].

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 [10]. 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 if done manually, spurring the creation of frameworks that facilitate automatic performance tuning, or *auto-tuning*, to automatically tune GPU applications [1,14,6,15,18].

As a consequence of the widespread adoption of GPUs for computation, increased complexity of GPUs, and improvements in auto-tuning, the number of parameters to be tuned is increasing, as well as the range of values per parameter. This leads to a large number of possible combinations and is reflected in auto-tuned GPU applications, resulting in search spaces with millions of configurations in practice [11,8,9].

The dramatically increased search space size creates new challenges for auto-tuning GPU applications, as auto-tuners have to increasingly rely on *optimization algorithms* to efficiently navigate the vast space of possible implementations. However, the performance of these optimization algorithms in large part depends on so-called hyperparameters, parameters of the optimization algorithms themselves. Optimizing these hyperparameters involves repeatedly auto-tuning a wide variety of search spaces, consisting of multiple kernels and inputs on various GPUs, for each hyperparameter configuration, with the goal of finding the best hyperparameter configuration for the general auto-tuning problem.

Despite the potential benefits of hyperparameter tuning optimization algorithms for auto-tuning problems, to the best of our knowledge, no other auto-tuning framework has applied extensive generalized hyperparameter tuning. This could be the case because it is not as straightforward as it might appear; hyperparameter tuning for optimization algorithms in GPU auto-tuning differs significantly from general hyperparameter tuning due to the nature of the objective, evaluation cost and scale, and hardware considerations. First, traditional hyperparameter tuning is largely agnostic to the underlying hardware, as long as the model can be trained. In contrast, hardware plays a crucial role in GPU auto-tuning, and optimization algorithms must be optimized across a variety of GPU architectures to achieve good generalization, adding an additional dimension and cost to the hyperparameter tuning problem. In addition, unlike traditional hyperparameter tuning which relies on the abstract number of function evaluations as a measure of time, GPU auto-tuning must consider the true time spent as perceived by the user, as the auto-tuning itself directly affects the amount of time spent. The resulting complex performance data must be aggregated accross iterations and problems, and translated to a single measure of general performance. Moreover, there are substantial differences in the cost and scale of evaluations; instead of training a model for hours or weeks, GPU auto-tuning requires compiling and executing kernels, typically taking at most seconds per evaluation and minutes for an optimization algorithm to reach a near-optimal solution. However, the distribution and structure of a search space 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 scalability challenges. 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. More specifically, we propose a systematic and general approach to hyperparameter optimization by building on the statistically robust methodology for comparing optimization algorithm performance for auto-tuning [21]. In addition, we introduce a novel *simulation mode* for simulating optimization algorithms for auto-tuning to make hyperparameter optimization feasible. Moreover, we promote the creation and use of reproducible and shareable resources on which the hyperparameter tuning is performed, democratizing hyperparameter tuning. Our contributions have been implemented in Kernel Tuner [18,20], an open-source Python tool 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 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].

Table 1 outlines various auto-tuning frameworks and features regarding optimization algorithms and hyperparameters. 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. This is consistent with our observation that authors of auto-tuning works have so far not prioritized hyperparameter tuning.

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 [14]	✓	✗	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 [15]	✓	✓	Simulated Annealing, Differential Evolution, Multi-armed Bandit, Local Search	✗
BaCO [9]	✓	✗	Bayesian Optimization, Exhaustive	✓(File-based)
Kernel Tuner [18]	✓	✓	20+ different global and local optimization algorithms, including Annealing methods, Genetic/Evolutionary methods, Swarm-based methods, and Bayesian Optimization	✓(API-based)

The most closely related work is perhaps [17], in which a limited hyperparameter tuning is conducted to fairly compare optimization algorithms for GPU auto-tuning. They test the hyperparameters on a combination of three kernels accross nine GPUs. Their approach illustrates the challenge in auto-tuning of representing the relation between time and optimization algorithm performance, as they create two bins for budgets based on function evaluations, do binary head-to-head comparisons between optimization algorithms, and select hyperparameter tuning configurations by repeatedly combining combinations from individual problems.

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:

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

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 a search space. In addition, the GPU auto-tuning search spaces are generally discontinuous and non-convex, requiring optimization algorithms suitable for this context to navigate the search spaces. Poorly tuned search strategies may result in 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 hyperparmeter 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 [21], provides a systematic approach to comparing search strategies across search spaces. It defines a performance metric $\mathcal{P}$ which quantifies an optimization algorithms' performance over the passed time relative to a calculated random search baseline accross search spaces. The hyperparameter tuning problem can be formalized as:

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

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 test set). The usage of $\mathcal{P}$ provides a statistically robust metric for hyperparameter tuning, enabling optimizing the optimization algorithm performance accross 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 ran 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 repeated 5 times, on 3 kernels accross 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 ~ 11.6 days, or ~ 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 52 compute days. At first glance, the brute-forcing does not seem sensible, however, 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 tuning execution is reproducible and free from external system noise. Finally, it reduces computational and energy cost and limitations: once the full search space is evaluated, hyperparameter tuning can be performed without the need for costly compilation 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 systematically 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. 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 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 [18]. It supports CUDA, HIP [13], 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 [17]. The wide variety of implemented optimization algorithms provide a large potential gain in performance by hyperparameter tuning, making Kernel Tuner an appropriate candidate for implementing our hyperparameter tuning method.

To summarize our hyperparameter tuning pipeline, the hyperparameter tuning we implemented in Kernel Tuner uses the autotuning methodology [21] software package to request a performance score for an optimization alorithm with a

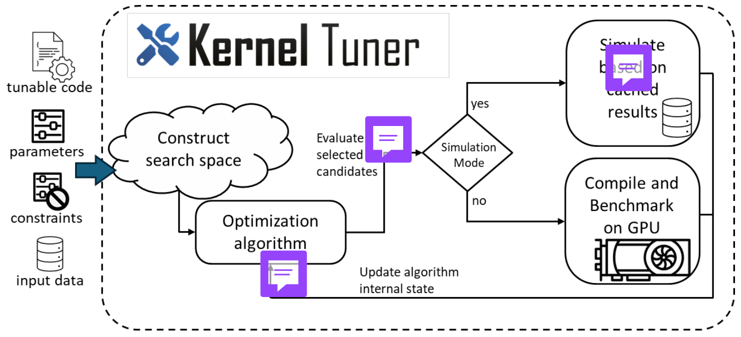

tunable code
parameters
constraints
input data
Kernel Tuner
Construct search space
Optimization algorithm
Evaluate selected candidates
Simulation Mode
yes
no
Simulate based on cached results
Compile and Benchmark on GPU
Update algorithm internal state

Fig. 1: Kernel Tuner extended with the option to simulate GPU auto-tuning.

specifc hyperparameter configuration accross the selected search spaces. The autotuning methodology package in turn uses the Kernel Tuner simulation mode to run the optimization algorithm with the hyperparameter configuration on each of the search spaces.

The simulation mode is implemented by extending Kernel Tuner's existing checkpointing mechanism. Figure 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, and the pre-recorded execution times (including compilation and benchmarking times) are used to update the tuning budget accordingly. From the point of view of the optimization algorithm, there is no perceivable difference between live tuning or with 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 extend 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 the T1 input and T4 output formats standards which have been formed by the auto-tuning community, to enable anyone to easily apply our method for scalable general hyperparameter tuning.

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

Algorithm	Hyperparameter	Values
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}
Greedy ILS	neighbor	{Hamming, **adjacent**}
	restart	{**True**, False}
	no_improvement	{10, **25**, 50, 75}
	random_walk	{**0.1**, 0.2, 0.3, 0.4, 0.5}
Simulated Annealing	T	{0.5, **1.0**, 1.5}
	T_min	{0.0001, **0.001**, 0.01}
	alpha	{0.9925, **0.995**, 0.9975}
	maxiter	{**1**, 2, 3}

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 [13]. The four GPUs are an AMD MI250X (in the LUMI supercomputer), an Nvidia A6000 (in the DAS VU cluster), an AMD W6600, an Nvidia A4000, and an Nvidia A100 (the latter three in the DAS ASTRON cluster). To demonstrate an important advantage of our hyperparameter tuning method, each application is fully brute-force auto-tuned on each GPU system, after which all other all evaluations are performed on the sixth generation DAS VU-cluster [3] using a Nvidia A4000 GPU node. This GPU is paired with a 24-core AMD EPYC-2 7402P CPU, 128 GB of memory, and 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.

The selected optimization algorithms and hyperparameters are shown in Table 2. A sensitivity test of the hyperparameters using the non-parametric Kruskal-Wallis test and mutual information scoring revealed that the W hyperparameter of PSO has no meaningful effect on the score, and as such is left out.

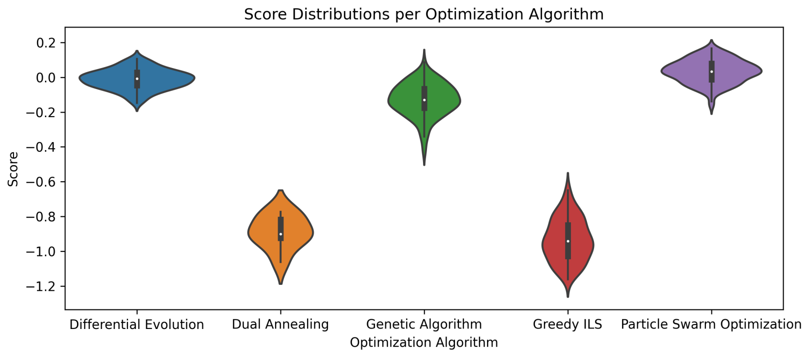


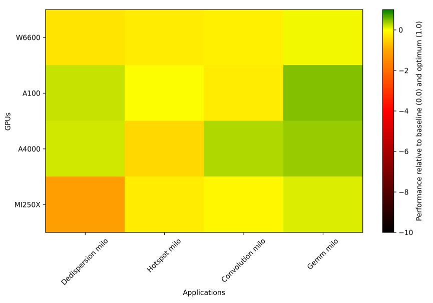
Fig. 2: Violin plots of the performance scores for all hyperparameter configurations of the evaluated optimization algorithms.

Each combination of hyperparameter values shown in Table 2 is ran 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, in accordance with the methodology [21]. In the performance comparisons of Section 4.2, each instance is measured with 100 repeats to mitigate stochasticity.

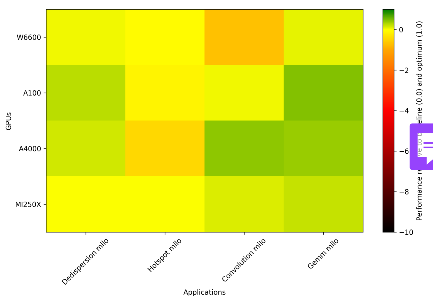
4.2 Results

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, an algorithm with a score of 0 performs as well as the random search baseline, while a score of 1 indicates that the optimum is found. When comparing two scores directly, a difference of 10% in the score can mean that given the same time, the algorithm with the higher score finds configurations that are 10% closer to the optimum. For more information on the performance scores, see [21]. The violin plots in Fig. 2 depict large differences in scores between the various hyperparameter configurations within an optimization algorithm, with an average improvement of 0.380, highlighting the potential performance improvement gained by hyperparameter tuning.

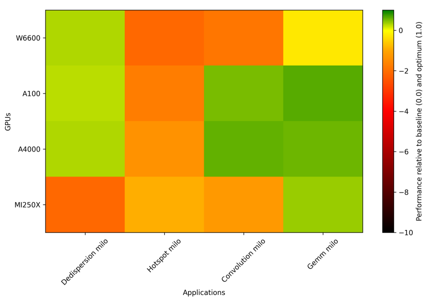
Next, we examine the overall performance and generalization of the hyperparameter tuning accross the search spaces in Fig. 3. Visual inspection of both the *untuned* (where the hyperparameters have been optimized for worst performance instead of best) and tuned for each of the optimization algorithms reveals that each of the tuned versions improve upon their counterpart in general, instead of over-optimizing on a limited number of search spaces to boost the score. In particular, it can be seen that performance is improved in both the applications



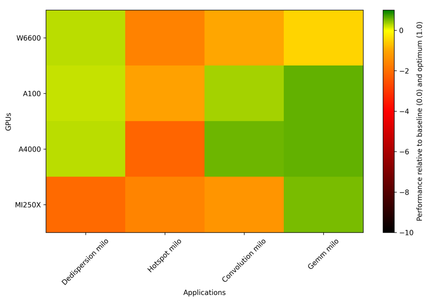
(a) Genetic Algorithm untuned



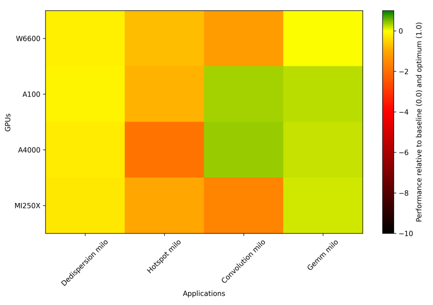
(b) Genetic Algorithm tuned



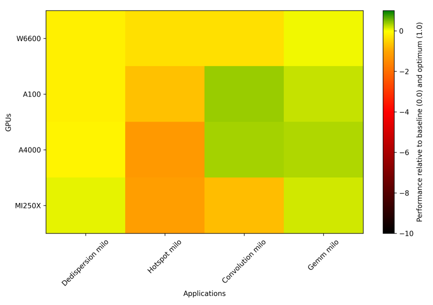
(c) Greedy ILS untuned



(d) Greedy ILS tuned



(e) Simulated Annealing untuned



(f) Simulated Annealing tuned

Fig. 3: Impact of tuning on optimization algorithm performance per search space.

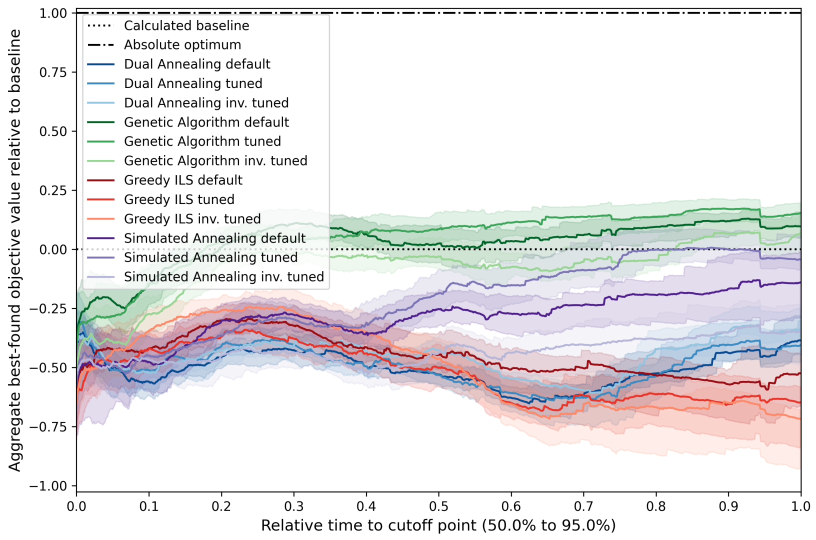


Fig. 4: Aggregate performance over time accross all search spaces.

tuned on (*dedispersion* and *hotspot*) as well as the applications that have not been tuned on (*convolution* and *GEMM*).

Moreover, the aggregate general performance is shown in Fig. 4, where it can be seen that over the course of time the tuned optimization algorithms generally perform best. Quantifying this on the test search spaces, Genetic Algorithm is improved by 0.136, Greedy ILS is improved by 0.051, and Simulated Annealing is improved by 0.252. This demonstrates that our hyperparameter tuning method exhibits good generalized performance.

Finally, as to the scalability of our method, we can compare it to the estimated times without the simulation mode. With our method, hyperparameter tuning took approximately 4.5 hours for the optimization algorithm with the least hyperparameter configurations and X hours for the optimization algorithm with the most hyperparameter configurations. 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 26 hours for the optimization algorithm with the least hyperparameter configurations and 347 hours for the optimization algorithm with the most hyperparameter configurations.

Set GA tuning time once done

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 and Availability

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 becomes increasingly crucial to fully leverage the computational power of modern GPUs. 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 navigating 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 achieve improved performance in finding near-optimal GPU kernel configurations. The introduction of a simulation mode significantly reduces the computational cost of hyperparameter tuning, making large-scale experiments feasible and preventing the need for constant access to hardware and excessive resource usage. By addressing the 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. Kernel Tuner is open source software, and contributions are welcome. For more information, please visit the GitHub repository[4].

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

References

1. Ansel, J., Kamil, S., Veeramachaneni, K., Ragan-Kelley, J., Bosboom, J., O'Reilly, U.M., Amarasinghe, S.: OpenTuner: An extensible framework for program autotuning. 2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT) pp. 303–315 (2014). https://doi.org/10.1145/2628071.2628092, http://groups.csail.mit.edu/commit/papers/2014/ansel-pact14-opentuner.pdf

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

2. Ashouri, A.H., Killian, W., Cavazos, J., Palermo, G., Silvano, C.: A Survey on Compiler Autotuning using Machine Learning. ACM Computing Surveys **51**(5), 1–42 (Sep 2019). https://doi.org/10.1145/3197978, https://dl.acm.org/doi/10.1145/3197978
3. Bal, H., Epema, D., de Laat, C., van Nieuwpoort, R., Romein, J., Seinstra, F., Snoek, C., Wijshoff, H.: A medium-scale distributed system for computer science research: Infrastructure for the long term. Computer **49**(05), 54–63 (May 2016). https://doi.org/10.1109/MC.2016.127, place: Los Alamitos, CA, USA Publisher: IEEE Computer Society
4. Balaprakash, P., Dongarra, J., Gamblin, T., Hall, M., Hollingsworth, J.K., Norris, B., Vuduc, R.: Autotuning in High-Performance Computing Applications. Proceedings of the IEEE **106**(11), 2068–2083 (Nov 2018). https://doi.org/10.1109/JPROC.2018.2841200, https://ieeexplore.ieee.org/document/8423171/
5. Falch, T.L., Elster, A.C.: Machine learning based auto-tuning for enhanced OpenCL performance portability. In: 2015 IEEE international parallel and distributed processing symposium workshop. pp. 1231–1240. IEEE, Hyderabad, India (2015). https://doi.org/10.1109/IPDPSW.2015.85
6. Filipovič, J., Petrovič, F., Benkner, S.: Autotuning of OpenCL kernels with global optimizations. In: Proceedings of the 1st workshop on AutotuniNg and adaptivity AppRoaches for energy efficient HPC systems. Andare '17, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3152821.3152877, https://doi.org/10.1145/3152821.3152877, number of pages: 6 Place: Portland, OR, USA tex.articleno: 2
7. Heldens, S., Hijma, P., Werkhoven, B.V., Maassen, J., Belloum, A.S.Z., Van Nieuwpoort, R.V.: The Landscape of Exascale Research: A Data-Driven Literature Analysis. ACM Computing Surveys **53**(2), 1–43 (Mar 2021). https://doi.org/10.1145/3372390, https://dl.acm.org/doi/10.1145/3372390
8. Heldens, S., van Werkhoven, B.: Kernel Launcher: C++ Library for Optimal-Performance Portable CUDA Applications (Mar 2023). https://doi.org/10.48550/arXiv.2303.12374, http://arxiv.org/abs/2303.12374, arXiv:2303.12374 [cs]
9. Hellsten, E.O., Souza, A., Lenfers, J., Lacouture, R., Hsu, O., Ejjeh, A., Kjolstad, F., Steuwer, M., Olukotun, K., Nardi, L.: BaCO: a fast and portable bayesian compiler optimization framework. In: Proceedings of the 28th ACM international conference on architectural support for programming languages and operating systems, volume 4. pp. 19–42. Asplos '23, Association for Computing Machinery, New York, NY, USA (Feb 2024). https://doi.org/10.1145/3623278.3624770, https://doi.org/10.1145/3623278.3624770, number of pages: 24 Place: Vancouver, BC, Canada
10. Hijma, P., Heldens, S., Sclocco, A., Van Werkhoven, B., Bal, H.E.: Optimization Techniques for GPU Programming. ACM Computing Surveys **55**(11), 1–81 (Nov 2023). https://doi.org/10.1145/3570638, https://dl.acm.org/doi/10.1145/3570638
11. Jacob O. Tørring, Ben van Werkhoven, Filip Petrovič, Floris-Jan Willemsen, Jiří Filipovič, Anne C. Elster: Towards a Benchmarking Suite for Kernel-tuners (accepted at iWAPT 2023) (Jan 2023), https://www.overleaf.com/project/638e0716ca3dc21d79f564ba
12. LeCun, Y., Bengio, Y., Hinton, G.: Deep learning. Nature **521**(7553), 436–444 (May 2015). https://doi.org/10.1038/nature14539, https://www.nature.com/articles/nature14539
13. Lurati, M., Heldens, S., Sclocco, A., Van Werkhoven, B.: Bringing Auto-Tuning to HIP: Analysis of Tuning Impact and Difficulty on AMD and Nvidia GPUs. In:

Carretero, J., Shende, S., Garcia-Blas, J., Brandic, I., Olcoz, K., Schreiber, M. (eds.) Euro-Par 2024: Parallel Processing, vol. 14801, pp. 91–106. Springer Nature Switzerland, Cham (2024). https://doi.org/10.1007/978-3-031-69577-3_7, https://link.springer.com/10.1007/978-3-031-69577-3_7, series Title: Lecture Notes in Computer Science

14. Nugteren, C., Codreanu, V.: CLTune: A generic auto-tuner for OpenCL kernels. 2015 IEEE 9th International ... (2015), https://ieeexplore.ieee.org/abstract/document/7328205/, publisher: IEEE
15. Rasch, A., Gorlatch, S.: ATF: A generic directive-based auto-tuning framework. Concurrency and Computation: Practice and Experience (2018), https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4423, publisher: Wiley Online Library
16. Schoonhoven, R., Veenboer, B., Van Werkhoven, B., Batenburg, K.J.: Going green: optimizing GPUs for energy efficiency through model-steered auto-tuning. 2022 IEEE/ACM International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS) pp. 48–59 (Nov 2022). https://doi.org/10.1109/PMBS56514.2022.00010, https://ieeexplore.ieee.org/document/10024022/, conference Name: 2022 IEEE/ACM International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS) ISBN: 9781665451857 Place: Dallas, TX, USA Publisher: IEEE
17. Schoonhoven, R., van Werkhoven, B., Batenburg, K.J.: Benchmarking optimization algorithms for auto-tuning GPU kernels. IEEE Transactions on Evolutionary Computation pp. 1–1 (2022). https://doi.org/10.1109/TEVC.2022.3210654, http://arxiv.org/abs/2210.01465, arXiv:2210.01465 [cs]
18. van Werkhoven, B.: Kernel Tuner: A search-optimizing GPU code auto-tuner. Future Generation Computer Systems **90**, 347–358 (Jan 2019). https://doi.org/10.1016/j.future.2018.08.004, https://www.sciencedirect.com/science/article/pii/S0167739X18313359
19. van Werkhoven, B., Palenstijn, W.J., Sclocco, A.: Lessons learned in a decade of research software engineering gpu applications. In: International conference on computational science. pp. 399–412. Springer (2020)
20. Willemsen, F.J., van Nieuwpoort, R., van Werkhoven, B.: Bayesian Optimization for auto-tuning GPU kernels. In: 2021 International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS). pp. 106–117 (Nov 2021). https://doi.org/10.1109/PMBS54543.2021.00017
21. Willemsen, F.J., Schoonhoven, R., Filipovič, J., Tørring, J.O., van Nieuwpoort, R., van Werkhoven, B.: A methodology for comparing optimization algorithms for auto-tuning. Future Generation Computer Systems **159**, 489–504 (2024). https://doi.org/https://doi.org/10.1016/j.future.2024.05.021, https://www.sciencedirect.com/science/article/pii/S0167739X24002498