Tuning the tuner: the art of automatic performance optimization

Willemsen, F.Q.

Tuning the Tuner: The Art of Automatic Performance Optimization

Proefschrift

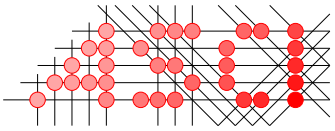
ter verkrijging van
de graad van doctor aan de Universiteit Leiden,
op gezag van rector magnificus prof.dr. S. de Rijcke,
volgens besluit van het college voor promoties
te verdedigen op dinsdag 14 april 2026
klokke 14:30 uur

door

Floris-Jan Quirinus Willemsen
geboren te Middelharnis, Nederland
in 1997

Promotor:	Prof. dr. R.V. van Nieuwpoort	
Co-promotor:	Dr. B. van Werkhoven	
Promotiecommissie:	Prof. dr. T.H.W. Bäck	
	Prof. dr. M.M. Bonsangue	
	Prof. dr. A. Plaat	
	Prof. dr. A.D. Pimentel	Universiteit van Amsterdam
	Prof. dr. ir. A.L. Varbanescu	Universiteit Twente
	Prof. dr. R. Vuduc	Georgia Institute of Technology

Universiteit Leiden



Advanced School for Computing and Imaging

Printed by: Optima Grafische Communicatie

ISBN 978-94-6534-280-1

This dissertation is available online at https://scholarlypublications.universiteitleiden.nl/

This work was carried out in the ASCI graduate school. ASCI dissertation series number 483.

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).
The ESiWACE3 project has received funding from the European High Performance Computing Joint Undertaking (JU) under grant agreement No 101093054. We acknowledge the EuroHPC Joint Undertaking for awarding this project access to the EuroHPC supercomputer LUMI, hosted by CSC (Finland) and the LUMI consortium through a EuroHPC Regular Access call..

“I want to take everything I’ve seen and thought and learned
and reduce them and relate them and refine them
until I have something of meaning, something of use.”

John Steinbeck, *Sweet Thursday*, 1954

Table of Contents

1 Introduction

Modern High-Performance Computing (HPC) systems increasingly rely on heterogeneous architectures, combining multicore CPUs with accelerators such as Graphics Processing Units (GPUs). These platforms enable major advancements in fields including artificial intelligence, climate modeling, and radio astronomy, but achieving high performance remains challenging. Application performance depends on numerous interdependent implementation choices, such as parallelization strategies, data layouts, and algorithmic variants, whose effects on performance are often non-linear, hardware-dependent, and hard to predict. The combinatorial explosion of possible parameter configurations, coupled with hardware and software constraints, makes manual tuning infeasible.

Auto-tuning addresses this challenge by automating the search for high-performing configurations. Auto-tuners systematically generate and evaluate program variants, varying performance-critical parameters to optimize a chosen metric such as execution time or energy efficiency. While effective, auto-tuning introduces a new optimization problem: how to design and configure the auto-tuning process itself. Choices such as search space construction, optimization algorithm evaluation and selection, and constraint handling have a substantial impact on tuning efficiency and results.

This thesis investigates the *optimization of auto-tuning problems* with a focus on GPU kernel optimization and HPC workloads. It develops methods to make auto-tuning more efficient, scalable, and robust.

Thesis Statement: The central aim of this thesis is to advance the efficiency, scalability, and robustness of auto-tuning by establishing methods for evaluating and improving optimization algorithms, search space construction, hyperparameter tuning, and constraint handling in the context of GPU and HPC applications.

1

1.1 Auto-tuning

High-Performance Computing (HPC) has become a cornerstone for advancing scientific discovery and industrial innovation, powering applications in domains such as climate modeling, artificial intelligence, quantum physics, fluid dynamics, and radio astronomy [66, 87, 123, 110]. The massive parallelism of GPUs and the increasing integration and reliance of modern HPC systems on GPUs has enabled breakthroughs across disciplines. However, fully exploiting the computational potential of these platforms remains challenging [167, 70].

Programming and optimizing for GPUs and other accelerators involves making numerous interdependent design decisions that have a complex and often unpredictable impact on performance. These decisions include mapping computations to parallel threads and thread blocks, selecting algorithms and data structures, organizing data in different memory hierarchies, and applying code transformations such as loop tiling, vectorization, or unrolling. Many of these choices are governed by discrete parameters with a wide range of valid values, such as block sizes, tile dimensions, and data layouts. Moreover, a large fraction of the possible parameter combinations can be infeasible due to hardware limits, e.g., exceeding shared memory capacity, register file size, or other architectural constraints, or implementation-specific limitations.

The performance optimization process is therefore inherently combinatorial: the set of all possible parameter combinations forms a vast, discrete search space. Only a small fraction of configurations typically yield near-optimal performance [136, 134, 165]. Manually exploring these spaces is infeasible due to their size, the complexity of hardware–software interactions, and the cost of compiling and executing each candidate configuration.

Auto-tuning addresses this challenge by automating the search for high-performing configurations [9]. Auto-tuning frameworks systematically generate and evaluate program variants by varying tunable parameters; performance-critical parameters that govern aspects such as thread geometry, work distribution, memory layout, and algorithmic strategies. Evaluation typically involves compiling the variant (if necessary), executing it on the target hardware, measuring relevant metrics such as execution time or energy consumption, and validating correctness.

Over the years, auto-tuning has been successfully applied in a variety of domains, yielding high-performance libraries such as FFTW for Fast Fourier Transforms [50] and ATLAS for linear algebra [168], as well as specialized frameworks for domains including image processing [101, 163], fluid dynamics [44], and radio astronomy [137]. General-purpose auto-tuning frameworks [3, 114, 125, 165] provide application-independent mech-

anisms for defining tunable parameters and systematically exploring the resulting search spaces.

Since exhaustive exploration is rarely feasible, auto-tuners often employ optimization algorithms to guide the search efficiently. These algorithms aim to identify high-performing configurations within acceptable time budgets, balancing exploration of the search space with exploitation of promising regions. Depending on the application, the optimization objective may be to maximize performance, minimize energy consumption, or optimize a combination of metrics.

In short, auto-tuning represents a critical intersection of performance engineering, empirical measurement, and combinatorial optimization. It provides a systematic way to bridge the gap between the theoretical capabilities of modern computing hardware and the practical performance achieved by applications, making it a fundamental tool in the HPC optimization toolkit.

1.2 Research Questions

While auto-tuning has emerged as a powerful method to automate performance optimization, several fundamental challenges remain unresolved. First, the search spaces defined by tunable parameters are vast, highly discrete, and often constrained, making exhaustive exploration infeasible. Many parameter configurations are invalid due to hardware or implementation limitations, and current search strategies struggle to balance efficiency with robustness in such irregular domains.

Second, the optimization algorithms that drive auto-tuning are difficult to evaluate and compare in a reproducible manner. Studies often report results under differing conditions, budgets, and metrics, preventing meaningful comparisons between methods. Without a common methodology, the community lacks a systematic way to assess algorithmic advances.

Third, constructing search spaces themselves can become a scalability bottleneck. For applications with large search space and complex constraints, resolving the search space may take minutes to days, limiting the practicality of auto-tuning at larger problem scales.

Fourth, while optimization algorithms have hyperparameters that strongly influence their behavior, these hyperparameters are rarely tuned in practice. This neglect leaves significant performance improvements untapped and obscures our understanding of how to configure optimizers effectively.

Fifth, existing optimization methods often waste effort evaluating infeasible configurations. Although constraint-aware techniques are common in related fields, they have yet to be systematically developed and applied in the context of auto-tuning.

1

Finally, recent advances in Large Language Models (LLMs) raise the possibility of automatically generating bespoke optimization algorithms, tailored to the characteristics of individual search spaces and applications. However, it is unclear whether LLMs can produce optimizers that are both effective and robust in an auto-tuning setting, and whether incorporating knowledge of the search space into their design process can provide a tangible advantage over human-crafted approaches.

The work presented in this thesis is guided by the following question:
How can we design, evaluate, and enhance optimization methods to make auto-tuning GPU kernels more effective, efficient, and adaptable across diverse problem settings? This is, in turn, addressed in the following research questions:

RQ1 (Chapter 2) Can Bayesian optimization be applied for more effective performance tuning of GPU kernels?

RQ2 (Chapter 3) How can optimization algorithms for auto-tuning be compared in a fair and reproducible manner?

RQ3 (Chapter 4) How can large search spaces for auto-tuning be constructed efficiently while minimizing infeasible configurations?

RQ4 (Chapter 5) Can hyperparameter optimization be used to improve the effectiveness of auto-tuners?

RQ5 (Chapter 6) Can constrained optimization techniques be applied to handle infeasible configurations in auto-tuning problems?

RQ6 (Chapter 7) Can Large Language Models, when guided by problem-specific knowledge, generate effective bespoke optimization algorithms for auto-tuning?

1.3 Main Contributions of this Thesis

[170] F.-J. Willemsen, R. van Nieuwpoort, and B. van Werkhoven. "Bayesian Optimization for auto-tuning GPU kernels". In: *2021 International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS) at SuperComputing 2021*. 2021 International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS) at SuperComputing 2021. Nov. 2021, pp. 106–117.

[173] F.-J. Willemsen, R. Schoonhoven, J. Filipovič, J. O. Tørring, R. van Nieuwpoort, and B. van Werkhoven. "A methodology for comparing optimization algorithms for auto-tuning". In: *Future Generation Computer Systems* 159 (2024), pp. 489–504.

[159] J. O. Tørring, B. van Werkhoven, F. Petrovč, F.-J. Willemsen, J. Filipovič, and A. C. Elster. "Towards a benchmarking suite for kernel tuners". In: *2023 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*. IEEE, 2023, pp. 724–733.

[171] F.-J. Willemsen, R. V. van Nieuwpoort, and B. van Werkhoven. "Efficient construction of large search spaces for auto-tuning". In: *Proceedings of the 54th international conference on parallel processing (ICPP '25)*. San Diego, CA, USA and New York, NY, USA: Association for Computing Machinery, Sept. 2025 *(Best Paper Award)*.

[172] F.-J. Willemsen, R. V. van Nieuwpoort, and B. van Werkhoven. "Tuning the Tuner: Introducing Hyperparameter Optimization for Auto-Tuning". In: *Proceedings of the 2025 IEEE eScience conference*. 21st IEEE International Conference on eScience. Chicago, IL, USA, Sept. 2025.

[169] F.-J. Willemsen, S. Heldens, R. V. van Nieuwpoort, and B. van Werkhoven. "Constraint-aware Optimization in Auto-Tuning". In: *2026 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. New Orleans, LA, USA, 2026.

[174] F.-J. Willemsen, N. van Stein, and B. van Werkhoven. "Automated Algorithm Design for Auto-Tuning Optimizers". In: *Proceedings of Machine Learning and Systems*. MLSys 2026. Seattle, WA, USA, 2026.

2 Bayesian Optimization for auto-tuning GPU kernels

> *All models are wrong, but some are useful.*
>
> George Box, *Empirical model-building and response surfaces*, 1987

Tuning GPU kernels involves navigating large, discrete, and constrained search spaces, where evaluations are expensive and derivatives are unavailable. These properties make Bayesian Optimization (BO) a natural choice, though its application to this domain introduces unique challenges. We address these by introducing a contextual variance exploration factor, new scalable acquisition functions, and an informed selection mechanism for acquisition strategies. Our BO approach handles rough search spaces with many invalid configurations effectively. Evaluation on diverse GPU kernels shows that our methods generalize well and consistently outperform existing search strategies in Kernel Tuner and other BO implementations.

This chapter is based on "Bayesian Optimization for auto-tuning GPU kernels" by Willemsen, van Nieuwpoort, and van Werkhoven [170].

2.1 Introduction

The multitude of implementation choices and invalid configurations for GPU kernels result in very large, non-convex, and discontinuous kernel design spaces [133, 114, 143, 93].

Several researchers have suggested using machine learning or metaheuristics to speedup the search process [79, 139, 10, 3, 114, 49, 165]. However, several of these approaches have been shown to not outperform random search [114, 10], or do not have a clear winner that clearly outperforms other methods and reliably returns highly-efficient configurations for a wide range of different kernels and devices [49, 165]. Therefore, in this chapter, we explore the application of Bayesian Optimization (abbreviated to *BO*) within the problem domain of auto-tuning a GPU kernel to a particular GPU. BO is known as an optimization method capable of quickly and reliably finding the global optimum of functions with unknown derivatives.

In this chapter, we present the following contributions:

- We present an implementation of a Bayesian Optimization search strategy, with a customized search space representation, dynamic exploration factor, and novel acquisition functions, specifically designed for auto-tuning GPU kernels.
- We demonstrate that our implementation of BO outperforms other existing search strategies in Kernel Tuner [165] and existing widely-used BO frameworks by a wide margin. On average, our best implementation performs 49.7% better at finding parameter configurations than the best other search strategy in Kernel Tuner, and 75% better than the second-best.
- We integrate our BO implementation in the open source Kernel Tuner, making it directly available for use. Additionally, we extend Kernel Tuner with a simulation mode, to enable benchmarking of search strategies without the need for a GPU.

Section 2.2 discusses the related works. Section 2.3 describes the application and optimization of Bayesian Optimization to the problem domain. Section 2.4 presents our experimental results, and Section 2.5 concludes.

2.2 Related Work

From chocolate chip cookies [142] to COVID [11]: Bayesian Optimization has been applied in a wide range of subjects, including auto-tuning. To grasp the current state of BO in GPU auto-tuning, this section briefly explores related applications of BO in auto-tuning. For further reference, Shahriari, Swersky, Wang, Adams, and Freitas [140] includes a review of BO and possible applications, listing a wide range of areas where BO has been applied.

The parameters of high-performance libraries like Basic Linear Algebra Subroutines (*BLAS*) used to be manually tuned and hard-coded, and at runtime selected based on architecture and some input characteristics. Recent developments require a more dynamic approach. Cianfriglia, Vella, Nugteren, Lokhmotov, and Fursin [30] applies decision trees to tackle this problem, showing significant performance gains on NVIDIA and ARM SoC GPUs. BLAS-libraries offer a clearly defined problem, as opposed to the unknown kernels inserted by the user in our case. Nevertheless, this shows that predictive modelling methods can be useful for GPU auto-tuning.

Another example of application of a machine learning algorithm to auto-tuning is Adaptive Sampling, a form of Active Learning, in Adaptive Sampling Kit (*ASK*) [116]. Like BO, Adaptive Sampling revolves around optimizing the sampling process by focusing on areas of uncertainty. However, Active Learning in general differs from BO as it has different selection mechanisms and is most often used for classification [24].

In a recent study, Wu, Kruse, Balaprakash, Finkel, Hovland, Taylor, et al. [176] apply BO to a search space of LLVM Clang / Polly pragma configurations on the PolyBench Benchmark suite. Their method appears to be most successful with random forests. Although specifically targeting the Polybench benchmarks and CPU auto-tuning, this shows that optimization of discrete performance search spaces with BO can be successful.

An interesting approach is the notion of Structured Bayesian Optimization (*SBO*), in which expert knowledge (such as performance models) are combined with BO to give the algorithm an edge over generic BO. This concept of SBO made its debut in auto-tuning with a publication by Dalibard, Schaarschmidt, and Yoneki [36] concerning the BespOke Auto-Tuner (*BOAT*), which provides custom-tailored auto-tuners based on expert knowledge. Although this structured BO is shown to be effective and mitigates the reportedly poor performance of BO on a high number of dimensions, it is intended for clearly specified problems where expert knowledge can be obtained and applied (e.g. BLAS), and as such does not generalize to our problem domain.

Miyazaki, Sato, and Shimizu [106] apply BO to increase the energy efficiency of HPC systems. Their case concerns the tuning of a HPC system to improve its ranking on the GREEN 500 benchmark list. As such, the right balance between performance and energy consumption must be found. While their approach is geared specifically towards the GREEN 500 benchmark, their positive results show potential for the application of BO to a broader problem domain.

Finally, Liu, Sid-Lakhdar, Marques, Zhu, Meng, Demmel, et al. [95] propose GPTune, a framework specifically targeting exascale systems using modified Gaussian Processes for multi-task auto-tuning of applications, allowing for function evaluation in parallel.

Although GPTune focuses on exascale applications instead of GPU auto-tuning, it demonstrates the potential of parallel multi-objective BO on discrete search spaces.

While the Bayesian Optimization applications seen in this related works section relate to auto-tuning and approaches contain elements similar to our problem domain, to the best of our knowledge, BO has not yet been applied to GPU auto-tuning. The GPU auto-tuning problem domain is different from CPU auto-tuning. CPU auto-tuning focuses mostly on compiler flags, while GPU auto-tuning also deals with resource partitioning, strictly invalid configurations (configurations resulting in errors instead of a performance impact), and specific constraints (e.g., a fitting block size for matrix multiplication).

2.3 Design and Implementation

In this section, we describe the application of Bayesian Optimization to the problem domain of auto-tuning GPU kernels. We start with a brief description of BO in Section 2.3.1, and Section 2.3.7 describes our implementation.

2.3.1 Bayesian Optimization

Essentially Bayesian Optimization comprises an objective function, a surrogate model and an acquisition function. The objective function $f : \mathscr{X} \to \mathbb{R}$ is considered as an otherwise unknown (black box) function for which the global optimum is to be found. As the goal of auto-tuning is usually to minimize the usage of resources (such as energy consumption or time), the optimum will from here on be assumed to be a minimum. This global optimum is then expressed by $x' = \arg\min_{x \in \mathscr{X}} f(x)$, where $\mathscr{X}$ is the search space, consisting of all possible combinations of all parameter values. The goal of the *surrogate model* is to emulate the objective function, especially in the optimization direction. The *acquisition function* then optimizes the surrogate model to suggest the next parameter configuration for evaluation with the objective function. The resulting observation is fitted to the surrogate model, improving the likeness of the surrogate model to the objective function, making the next suggestion of the acquisition function more precise.

2.3.2 Gaussian processes & covariance functions

The surrogate model can be formed with various techniques, such as Gaussian processes, Tree Parzen Estimators [82], Support Vector Machines [124], or Random Forests [107]. Each of these has their advantages and disadvantages. SVMs do not offer a probabilistic interpretation, Gaussian processes are computationally intensive, Random Forests tend

to require more observations than Gaussian processes [107], and TPEs do not model the interaction between hyperparameters.

To form the surrogate model, we employ the commonly used Gaussian processes (*GP*), which consist of a mean function and a covariance function. This covariance (or kernel function) describes the covariance of the random variables in the Gaussian process. While the mean function is usually a zero function, the choice of a kernel for the covariance function can have a significant impact on the surrogate model.

Kernels come in several variations, such as linear and periodic kernels, which can be combined to form a custom kernel befitting a problem domain where there can be strong assumptions on the shape of the objective function given expert knowledge. A commonly used default kernel for BO is the Radial Basis Function (*RBF*, also known as the Squared Exponential kernel), given several properties that make it an adequate default: it is universal, every function in the prior has infinitely many derivatives and it has two parameters, the lengthscale l and signal or output variance σ^2 [127]. The lengthscale determines the smoothness of the function, whereas the output variance determines the scaling. A popular variant on the RBF kernel is the Radial Quadratic kernel, which is in essence a combination of RBFs with varying lengthscales, generally known as a *scale mixture*. An additional parameter α determines the weighting of variation in scales.

The Matérn kernel [145] can be seen as a generalization of RBF, which is more suitable for physical processes, as the Matérn kernel is finitely differentiable. In addition, Le and Smola [85] have shown the Matérn kernel to be more suitable for high-dimensional inputs compared to the RBF kernel, as it suffers less from *concentration of measurement* problems. The Matérn kernel is defined by Rasmussen and Williams [127] as $k(r) = \frac{2^{1-\nu}}{\Gamma(\nu)}(\frac{\sqrt{2\nu}r}{l})^{\nu}K_{\nu}(\frac{\sqrt{2\nu}r}{l})$, where K_{ν} is a modified Bessel function [1], and ν and l are positive parameters.

The Matérn kernel contains a specific subclass of kernels: when the parameter ν is a positive half-integer, the Matérn kernel is simplified to the product of an exponential and a polynomial. According to Rasmussen and Williams [127], for $\nu \leq \frac{1}{2}$, the resulting process is very rough, whereas for $\nu \geq \frac{7}{2}$, the process becomes too smooth to model physical processes, making it practically indistinguishable from RBF. As such, $\nu = \frac{3}{2}$ and $\nu = \frac{5}{2}$ are the most interesting instances, which can be simplified to the following kernel functions respectively:

$k_{\nu=\frac{3}{2}}(r) = \left(1 + \frac{\sqrt{3}r}{l}\right)\exp\left(-\frac{\sqrt{3}r}{l}\right)$, and

$k_{\nu=\frac{5}{2}}(r) = \left(1 + \frac{\sqrt{5}r}{l} + \frac{5r^2}{3l^2}\right)\exp\left(-\frac{\sqrt{5}r}{l}\right)$.

Choosing an appropriate covariance function is important to achieve good predictive performance within few evaluations [141]. For auto-tuning GPU kernels, we require a

generally applicable covariance function as the objective function depends on the user input, simple covariance functions, such as linear and periodic, or combinations of these are not suited. While both RBF and Radial Quadratic are often used as defaults, they may not be particularly suited to our problem. These kernels tend to work best on smooth functions. As the lengthscale is usually adjusted based on the least smooth part of the surrogate model, discontinuities in the search space can severely disrupt the model. We thus use a fixed lengthscale.

As such, the Matérn kernel, given the discrete method proposed, the rougher Matérn $\nu = \frac{3}{2}$ combined with a larger (> 1) lengthscale appears to be a good choice to generalize well, making for a rougher function with a smoothing lengthscale. Alternatively $\nu = \frac{5}{2}$ with a shorter lengthscale (< 1) may be a suitable choice as well, as a smoother function with a rougher lengthscale.

2.3.3 Acquisition functions

Having established a GP-surrogate model g, this must now be sequentially refined to approach the objective function f. As evaluation of f is expensive, there must be a clever approach to which points to refine. To obtain the most useful information from a single evaluation, either the most uncertain region or the most promising region in the surrogate model should be further evaluated, an exploration / exploitation trade-off. This is captured by the acquisition function, which determines which parameter configuration to evaluate next by optimizing the score assigned to the parameter configurations x based on the predictions of the surrogate model. The point to be evaluated is expressed by $x_t = \underset{x \in \mathscr{X}}{\arg\min}\, a(x)$, with an acquisition function a and number of observations t. This expression is similar to the global optimum x', the difference being that a uses the inexpensive to evaluate surrogate model g, while f is our expensive to evaluate objective function. With the candidate parameter configuration decided by the acquisition function, f is used to obtain the y_t observation: $y_t = f(x_t) + \phi_t$, with ϕ_t the noise in the observation at number of observations t. The optimal configuration can be extracted at any time: $x^+ = \underset{x_i \in \mathscr{X}_{1:t}}{\arg\min}\, f(x_i)$, with x_i the parameter combination evaluated at timestep i.

Three acquisition functions are commonly used, and will be used here as well: Probability of Improvement, Expected Improvement and Upper Confidence Bound.

Probability of improvement (*PI*) is a conceptually simple acquisition function, which selects the next point based on the likelihood that it results in an improvement over the current optimum, $f(x^+)$ [83]. PI is expressed as $a_{PI}(x) = P(g(x) \leq (f(x^+) + \lambda))$, with P the probability and a constant $\lambda \geq 0$ determining exploration / exploitation. Expected Improvement (*EI*), in contrast to PI, includes discrimination on the size of the poten-

tial improvement [108]. Expected improvement is expressed as $a_{EI}(x) = \mathbb{E}(\max(g(x) - f(x^+), 0))$, where $\mathbb{E}$ is the expected value. Another popular acquisition function, the Upper Confidence Bound (*UCB*), is fundamentally different from PI and EI, as it is optimistic in the face of uncertainty, whereas PI and EI are based on improvement. UCB is very simplistic in the basis: $a_{UCB}(x) = \mu(x) + \lambda\sigma(x)$, where λ is a non-negative parameter controlling exploration [140].

As we assume minimization, we use the minimization variants of these acquisition functions. The minimization variant of UCB is known as Lower Confidence Bound (*LCB*).

2.3.4 General structure

The GPU auto-tuning problem domain poses two main challenges for application of BO: the representations of parameter types and dealing with invalid configurations.

First, parameters can be of many types: integers, real numbers, strings, Booleans, etc. We must thus assume that there can be a mixture of parameter types. In contrast, most work on BO assumes continuous parameters in the form of real numbers. Traditionally, the solution to this problem has been to treat the search space as continuous and snap the continuous values to discrete values before evaluation. As described by Garrido-Merchán and Hernández-Lobato [54], an issue with this approach is the possible mismatch between the optimum given by the acquisition function and the actual set of parameter configurations, which can result in an inaccurate surrogate model and getting stuck on a single parameter configuration.

Solving these kinds of problems has recently received attention. Garrido-Merchán and Hernández-Lobato [54] propose a modified covariance function for integer and one-hot mapped categorical parameters, where the covariance is zero for impossible parameter values, effectively removing the distance between the discrete values. Similarly, Luong, Gupta, Nguyen, Rana, and Venkatesh [98] propose manipulation of the exploration factor of the acquisition function and the length scale of the covariance function to prevent the acquisition function from getting stuck. Daxberger, Makarova, Turchetta, and Krause [39] propose a mixed-variable BO approach named *MiVaBO*, using a linear surrogate model and Thompson sampling. Likewise, Swiler, Hough, Qian, Xu, Storlie, and Lee [151] propose several surrogate models for mixed discrete-continuous variables. Rubanova, Dohan, Swersky, and Murphy [131] observe that optimization of the acquisition function is more challenging for discrete search spaces, as gradient-based optimization can not be applied. They propose *Amortized Bayesian Optimization*, which is the usage of a generative model that can be used longer than a single acquisition function optimization. Ru, Alvi, Nguyen,

Osborne, and Roberts [130] propose a framework for multiple continuous and categorical inputs named *CoCaBO*, where multiple-armed bandits are used to select categorical inputs in addition to GP-based BO.

Non-linearity is another problem surrounding parameter values, as in many computer-related parameters are non-linear, such as powers of two. Given that these parameters can not be assumed to refer to spatial units such as with kriging, this leads to a distortion within the surrogate model due to the distances between values. To avoid this, we normalize all numerical inputs in a linear fashion, mapping them back to the original value before outputting.

Categorical values also pose a challenge when encoding. Simple integer encoding imposes an ordering on categorical values. Within machine learning, it is thus common practice to instead apply one-hot encoding in such situations, which would involve adding a binary dimension for every categorical value. Due to BO's unsuitability for high-dimensional problems, this is not an ideal solution. Instead, binary encoding could be a viable solution, given that it is much more efficient on the number of dimensions used compared to one-hot encoding, and does not suffer from the assumptions of integer encoding. However, a direct comparison does not seem to be in favor of binary encoding [138]. Due to the structure of Kernel Tuner inputs, binary encoding has not been applied, as the user is responsible for the ordering of the parameters.

Given the problems mentioned, it might seem odd to use a continuous surrogate model in the first place. However, as seen in Section 2.3.2, discrete surrogate models have their drawbacks as well, and as shown by Karlsson, Bliek, Verwer, and Weerdt [78], continuous surrogate models can be well-suited for optimization of discrete problems.

The second challenge is that of the invalid parameter configurations, creating points in the search space that should not be selected.

For GPU kernels, a configuration can be diagnosed to be invalid in three stages of the process: by checking the validity of individual parameters according to the programming model specification beforehand, via an error at compile time, and via an error at runtime. The latter can for example be due to an invalid configuration of thread blocks that are within programming model specification, but invalid on the actual GPU due to hardware restrictions, which is part of what makes this application specifically geared towards GPU kernels. The earlier invalid configurations can be detected, the better, as this avoids waste of time and computing resources for the user. To aid in this, Kernel Tuner allows users to impose restrictions on search spaces. However, we may not be able to detect all invalid configurations beforehand, and these thus need to be properly handled. This is more

challenging than it may seem at first glance. If we do not fit an observation for the invalid parameter configuration to the surrogate model, the same invalid parameter configuration will always be chosen by the acquisition function.

Setting an observation value of our choice (given that an invalid parameter configuration can not have an observation value), is not ideal either. For example, a simple solution might seem be setting the observed value for such invalid configuration in the surrogate model to the inverse of the absolute optimum (an uninformative absolute optimum being $-\infty$ when minimizing and ∞ when maximizing), so the acquisition function will not choose this parameter configuration again. Alternatively, the observed value could be set to the mean of all observations so far. As variance is an important factor in acquisition functions that are not purely exploiting, the invalid parameter configuration is unlikely to be chosen again by the acquisition function. Despite the fact that no information has been gained due to the invalid parameter configuration other than that it is invalid, both options distort the surrogate model in such a way that the acquisition function either gets stuck or is less likely to choose parameter configurations near an invalid parameter configuration. While it is possible that neighbouring configurations are also invalid, it is impossible to know in advance exactly which parameter configurations will be invalid, so affecting the surrogate model with an artificial observation value is undesirable.

To address both the parameter type and invalid configuration problems, we propose the following structure for the application of BO to the problem domain: a discrete, normalized search space, where the acquisition function is solely optimized on the non-evaluated parameter configurations. This avoids spatial representation issues and prevents the acquisition function from getting stuck. In addition, it allows invalid configurations to be ignored, without distorting the surrogate model to achieve this. As Kernel Tuner already reports an average over a user-defined number of runs per evaluation, there is little practical need to revisit evaluated parameter configurations.

2.3.5 Initial sampling

Given the black-box nature of BO, it is possible to start optimization on a completely unknown surrogate model. However, it is common to first take an initial sample of the search space to aid exploration. While random sampling is often used, it is likely that the samples are unevenly distributed throughout the search space. As such, Miyazaki, Sato, and Shimizu [106] suggest the use of Latin Hypercube Sampling (LHS). This technique ensures that the initial samples are spread evenly throughout the search space, providing a more balanced initial sample.

In practice, this can be made difficult due to the possibility of invalid configurations. As we want to explore large search space in as few evaluations as possible, a small Latin Hypercube Sample of a large search space is in principle more balanced than random sampling, but also more easily unbalanced when one or more samples are missing due to invalidity. To avoid this, we thus first apply Latin Hypercube Sampling. If there are invalid samples, these are replaced by random samples until all initial samples are valid. This combination avoids a skewed initial sample.

2.3.6 Dynamic exploration factor

As seen in Section 2.3.3, the acquisition functions have an exploration factor, which is often set to a constant value. However, it would intuitively seem more sensible to make an informed calculation for this hyperparameter based on the state of the surrogate model at every evaluation. This is what Jasrasaria and Pyzer-Knapp [76] proposed with *contextual improvement*, a variation on the probability of improvement using *contextual variance* for the λ exploration parameter, defined as $\frac{\overline{\sigma^2}}{f(x^+)}$, where $\overline{\sigma^2}$ is the mean of variances in the posterior surrogate model and $f(x^+)$ is the best found observation so far. The experiments by Lizotte [96] suggest that the constant $\lambda = 0.01$ works well in most cases, while counter-intuitively, setting a cooling schedule for the exploration factor does not work as well. This could mean contextual variance does not work well either. However, Jasrasaria and Pyzer-Knapp [76] demonstrate that their contextual improvement can outperform a constant factor, and in addition avoids dependence on this parameter. Hence, such an approach to exploration / exploitation can be of benefit given that we want to be our implementation to be as generally applicable as possible.

However, the contextual variance can not be applied as-is to our problem domain: it is intended for maximization, and behaves differently depending on the absolute scale of the observations. The latter means that given two maximization problems with a similar mean variance, for a problem with a lower maximum, exploration will be disproportionately large, whereas for a problem with a higher maximum, exploration will be practically ignored. This is undesirable behaviour, and as such, applying this contextual variance is not simply a matter of setting the function to the inverse to obtain the minimum, but requires a new function instead.

We propose a function that outputs a hyperparameter $0 \leq \lambda$, which scales in proportion to $f(x^+)$, the mean variance, and the initial sample mean. To overcome the observation size dependence, we use the initial sample mean μ_s to get the fraction of improvement of the initial sample mean to the current optimal. To normalize the hyperparameter, we assume 0 as the minimum and the mean of variances in the posterior surrogate model

after the initial sampling, $\overline{\sigma_s^2}$, as the maximum. This results in the following function: $\lambda = \left(\frac{\overline{\sigma^2}}{\mu_s / f(x^+)}\right) / \overline{\sigma_s^2}$.

2.3.7 Implementation

Several Python packages already implement BO, including *Bayesian Optimization* [113], *SciKit-Opt* [65] and *HyperOpt* [82]. However, we need to customize our BO implementation to the problem domain and therefore we create our own implementation on top of SciKit-learn's *Gaussian Process Regressor* [118].

While the existing implementations use techniques such as BFGS to optimize the acquisition function, due to the discrete structure of our search spaces this is not practical for our implementation. Instead, we exhaustively predict every discrete point in the model. To compensate, we reuse the predictions when using multiple acquisition functions.

We propose a '*multi*' acquisition function, which skips acquisition functions when they repeatedly suggest the same candidates. We register how many duplicates were suggested per acquisition function in the past iterations. When this count exceeds a threshold, the *skip threshold*, the conflicting acquisition functions are pitted against each other. The acquisition function with the lowest discounted observations score is kept, while the others will be skipped, encouraging efficiency. This discounted observation score is calculated per acquisition function by $dos_t = \sum_{i=1}^{t} o_i \cdot 0.9^{t-i}$, where t is the current iteration number, o is the vector of observations over time, and 0.9 is the *discount factor*. This gives more recent observations more weight. As the surrogate model becomes more accurate and the acquisition functions suggest the same candidates more often, gradually fewer acquisition functions will be used. Eventually a single acquisition function will remain, returning to the traditional approach of a single acquisition function.

In addition, we propose an '*advanced multi*' acquisition function similar to the '*multi*' acquisition function, but removing predictions to avoid duplicates and directly judging acquisition functions based on their discounted observations score instead. Invalid observations use the median of the valid observations in this score, to avoid skewing the score. If an acquisition function scores more than 5% above the mean of the discounted observations (assuming minimization) for *skip threshold* or more times, this acquisition function will be skipped and the counts of others reset. If an acquisition function scores more than 5% below the mean of the discounted observations for *skip threshold* or more times, this acquisition function will be promoted to be the only acquisition function for the remainder of the optimization. We will call this threshold of 5% in both cases the *required improvement factor*. This approach is intended to progressively select the optimal acquisition function for a specific problem.

The acquisition functions we propose are different from *GP-Hedge* [72], which attempts to select the optimal acquisition function per function evaluation, requiring full prediction and optimization of all acquisition functions at every function evaluation. Our methods require fewer predictions, evaluating acquisition functions in a round-robin fashion, optimizing one acquisition function per function evaluation, until the best-performing acquisition function remains.

2.3.8 Hyperparameter tuning

To further optimize the implementation and provide valuable defaults to Kernel Tuner users, we tune the hyperparameters of the initial sampling, surrogate model, and acquisition functions. The hyperparameters were tuned on the kernels described in Table 2.2 (see Section 2.4.1) on the Nvidia GTX Titan X GPU. The resulting optimal hyperparameters are displayed in Table 2.1.

Hyperparameter	**Best value(s)**
Covariance function lengthscale	3/2 2
Exploration factor	CV
Covariance function lengthscale (CV)	3/2 1.5
Skip threshold	5
Order of acquisition functions	(ei, poi, lcb)
Required improvement factor	0.1
Discount factor	0.65 (multi), 0.75 (advanced multi)
Initial sampling	maximin
Pruning	yes
Acquisition functions	advanced multi, multi, EI

Table 2.1: Result of hyperparameter optimization

2.4 Experimental results

In this section, we evaluate the performance of our BO implementation for auto-tuning GPU kernels. Section 2.4.1 describes the experimental setup. Section 2.4.2 compares our methods to the other Kernel Tuner methods, and Section 2.4.4 compares our methods to other BO frameworks. In addition, Sections 2.4.3 and 2.4.5 demonstrate how our methods generalize across GPUs and kernels respectively.

2.4.1 Methods

The search space depends on the tunable kernels as well as the hardware, hence we employ various GPUs to evaluate our search strategies. First is the NVIDIA GTX Titan X with the *Maxwell* architecture, which contains 3072 cores in 24 streaming multiprocessors

Kernel	Configurations	Invalid	Minimum	Tunable parameters
GEMM	17956	0 (0%)	28.307	M_{wg}, N_{wg}, K_{wg}, M_{dimC}, N_{dimC}, M_{dimA}, N_{dimB}, K_{WI}, V_{WM}, V_{WN}, S_{TRM}, S_{TRN}, S_A, S_B, PRECISION
Convolution	9400	3624 (38.5%)	1.625	filter_width, filter_height, block_size_x, block_size_y, tile_size_x, tile_size_y, use_padding, read_only
PnPoly	8184	323 (3.9%)	26.968	block_size_x, tile_size, between_method, use_precomputed_slopes, use_method

Table 2.2: Specifications of tunable kernels for the GTX Titan X. Minimum execution time is given in milliseconds.

with 12 GB GDDR5 memory [155]. We also evaluate the performance of our BO implementation on the NVIDIA RTX 2070 Super (2019, *Turing* architecture, 2560 cores, 40 SMs, 8 GB GDDR6 memory [156]) and the NVIDIA A100 (2020, *Ampere* architecture, 6912 cores, 108 SMs, 40 GB HBM2e memory [154]) in Section 2.4.3.

In this evaluation, we use three highly-tunable GPU kernels to evaluate and compare our optimization strategies, namely an OpenCL general dense matrix multiplication (*GEMM*) kernel, a *2D Convolution* kernel implemented in CUDA and a heterogeneous point-in-polygon (*PnPoly*) kernel. Their variation in programming model, type of calculation and tunable parameters are intended to serve as a diverse sample of GPU kernels.

- **GEMM.** The GEMM kernel is based on the tunable OpenCL GEMM kernel found in CLBlast [115]. GEMM has a large number of tunable parameters (15 in total), which describe the thread block dimensions, the square area on which each thread block operates, whether or not to use shared memory for either input matrix, and vector widths. The Cartesian product of these parameters has a size of 82944, the search space has a size of 17956 after application of restrictions, giving GEMM the largest constrained search space of all test kernels.

- **Convolution.** The final GPU kernel is a 2D Convolution kernel implemented in CUDA, specifically one for image filtering, an extension of the 2D convolution kernel presented in [166]. The tunable parameters describe the thread block dimensions, work per thread, and whether or not to use padding in shared memory to avoid shared memory bank conflicts and whether or not to use the read-only cache. The Cartesian product of these parameters has a size of 18432, after application of restrictions the search space contains 9400 parameter configurations.

- **PnPoly.** The Point-in-Polygon (PnPoly) kernel is taken from [58]. This is a heterogeneous kernel, where most of the computation happens on the GPU, but part of the computation calculation can be performed on the CPU (depending on a tunable parameter). The kernel overlaps CPU-GPU data transfers with computations on the GPU. As such, the CPU-GPU data transfers are also included in the run time. The other tunable parameters describe the thread block dimensions, work per thread, and switches between different algorithms for certain calculations. PnPoly has no restrictions applied, so the search space is a Cartesian product totalling 8184 configurations.

As a baseline for our experiments, every run of our BO search strategies uses 20 function evaluations for taking an initial sample and 200 function evaluations for the optimization process itself. Plots set off to the number of function evaluations will thus start at 20 and end at 220. Each run of the BO methods is repeated 35 times. The other search methods in Kernel Tuner and other frameworks used for comparison are repeated 35 times as well, while the random sample method is repeated 100 times due to the greater variance. For all plots lower is better unless otherwise noted.

To compare performance of an optimization methods over multiple kernels, we use the mean of the factor of deviation from the kernel mean absolute error mean, designated with *Mean Deviation Factor* (*MDF*). The mean absolute error (*MAE*) is calculated against the global minimum of the search space over the function evaluations from 40 to 220, as the first evaluations are relatively noisy and their results depend too much on the quality of the initial sample. More specifically, we calculate the mean absolute error by $\frac{1}{10}\sum_{i=2}^{11}|f(x^+_{20i}) - f(x')|$, with x' the globally optimal parameter configuration and x^+_{20i} the best found parameter configuration after $20i$ function evaluations. The mean MAE across the runs is used for the MDF. The mean deviation factor is used in the form of a bar chart, where the error bars show standard deviation of the factors. This MDF is calculated by taking the mean MAE for each kernel, dividing it by the mean of the mean MAE of all kernels. This quantifies the performance of a strategy, allowing for comparison of approaches across multiple kernels, while retaining differences in MAE without the influence of varying performance scales between kernels.

2.4.2 Comparison

We compare the performance of our Bayesian Optimization method for a number of selected acquisition functions with the other optimization methods in Kernel Tuner, as well as through an extended comparison based on how long it takes other search methods to achieve performance similar to our search strategies. The search methods in Kernel Tuner

GPU	Kernel	Search space size	Invalid	Minimum
RTX 2070 Super	GEMM	17956	0	17.112
	Convolution	7520	1744 (23.2%)	1.221
	PnPoly	8184	288 (3.5%)	12.325
A100	GEMM	17956	0	8.518
	Convolution	7520	1744 (23.2%)	0.739
	PnPoly	8184	317 (3.9%)	13.091

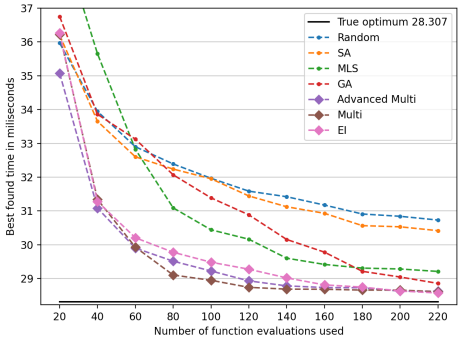
Table 2.3: Kernel details per GPU

selected for comparison are Simulated Annealing (*SA*), Multi-start Local Search (*MLS*) and Genetic Algorithm (*GA*), as these are the methods that performed best on the test kernels. The diamonds mark our methods, while the dots mark the other methods in Kernel Tuner. For all three GPU kernels in Figs. 2.1a to 2.1c, our methods appear to perform relatively similar. Most importantly, our search methods consistently outperform the other Kernel Tuner search methods. MLS and GA perform relatively well in the end, especially GA, but as seen in Fig. 2.1d, our methods perform better overall. These results appear to be promising: our methods have a much lower mean deviation factor than the other Kernel Tuner methods.

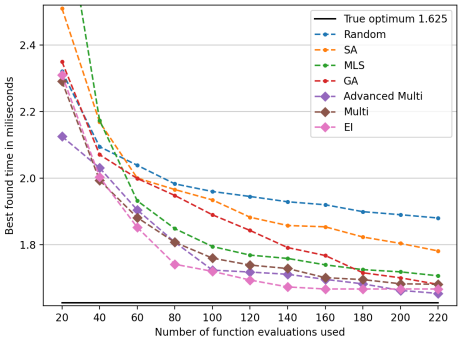
What has not been addressed is how hard or easy it is to find the optimum or near-optimum per search space; e.g. for some search spaces, it may be much harder to get to 95% of the optimum than to get to 90%. Due to space constraints it is not possible to show this for each combination of kernel and GPU, yet as an example we take GEMM on the GTX Titan X. Fig. 2.4 demonstrates the best found time of one of our acquisition functions (EI) at 220 function evaluations, and plots the other tuners at the number of function evaluations where they first match or exceed the best found time by EI, up to a maximum of 1020 function evaluations. In Fig. 2.1a at 220 unique function evaluations, GA appears to be relatively close to our methods. Yet Fig. 2.4 shows that it takes GA nearly 3 times as many unique function evaluations to match the best found time of EI. MLS performs slightly better, while SA and random do not get close to matching the best found time in five times as many function evaluations.

2.4.3 Other GPUs

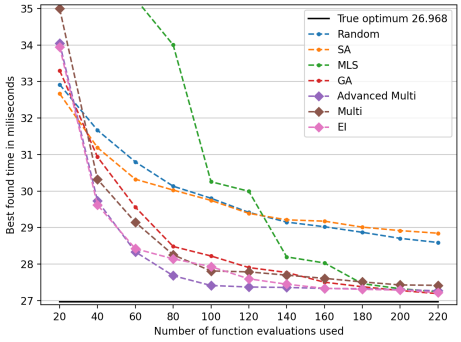
At this point, we have exclusively used the Nvidia GTX Titan X GPU to compare our results. To see how our search strategies fare on more modern, different GPUs, we compare results on the RTX 2070 Super and the A100. While the tunable kernel parameters are kept the same, the change of GPUs means that this is essentially a different search space, with different sizes, minima, and invalid configurations. These details are shown in Table 2.3.



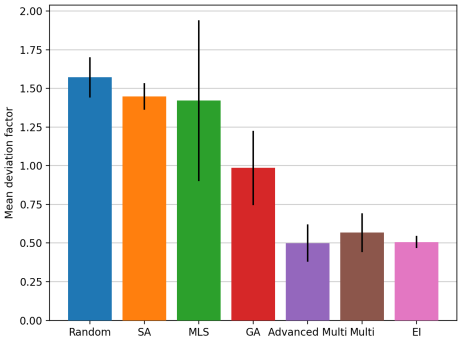
(a) GEMM



(b) Convolution



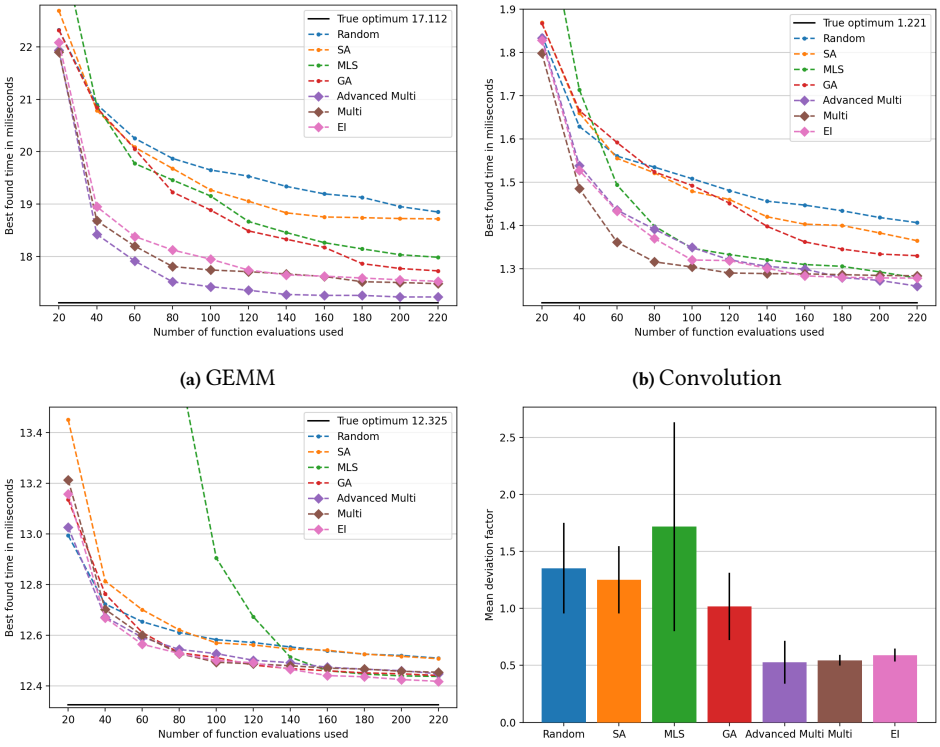
(c) PnPoly



(d) Mean deviation factors

Figure 2.1: Comparison of search methods on GPU kernels with the GTX Titan X; Figs. 2.1a to 2.1c show how the best found performance per method increases as the number of function evaluations increases.

The performance of the three GPU kernels on the RTX 2070 Super and A100 are shown in Figs. 2.2 and 2.3 respectively. For GEMM, our search strategies perform comparable to the GTX Titan X in both cases. The performance on Convolution is good in both cases as well. It must be noted that for Convolution on the A100, the first two best configurations are at 0.739 and 0.761, while the third best is at 0.823, explaining the large difference between the true optimum and the obtained performance in Fig. 2.3b. Also remarkable is the performance of the BO strategies for PnPoly on the A100, for which *advanced multi* and EI perform worse than for PnPoly on the GTX Titan X and the RTX 2070 Super. However, when combined with the *multi* acquisition function, BO is the only optimization method that is able to reliably return near-optimal configurations for PnPoly on the A100. Looking at Figs. 2.2d and 2.3d, we can see that our strategies generalize well to the kernels on other GPUs. The difference in the mean deviation factors between the A100 and GTX Titan X



(a) GEMM **(b)** Convolution

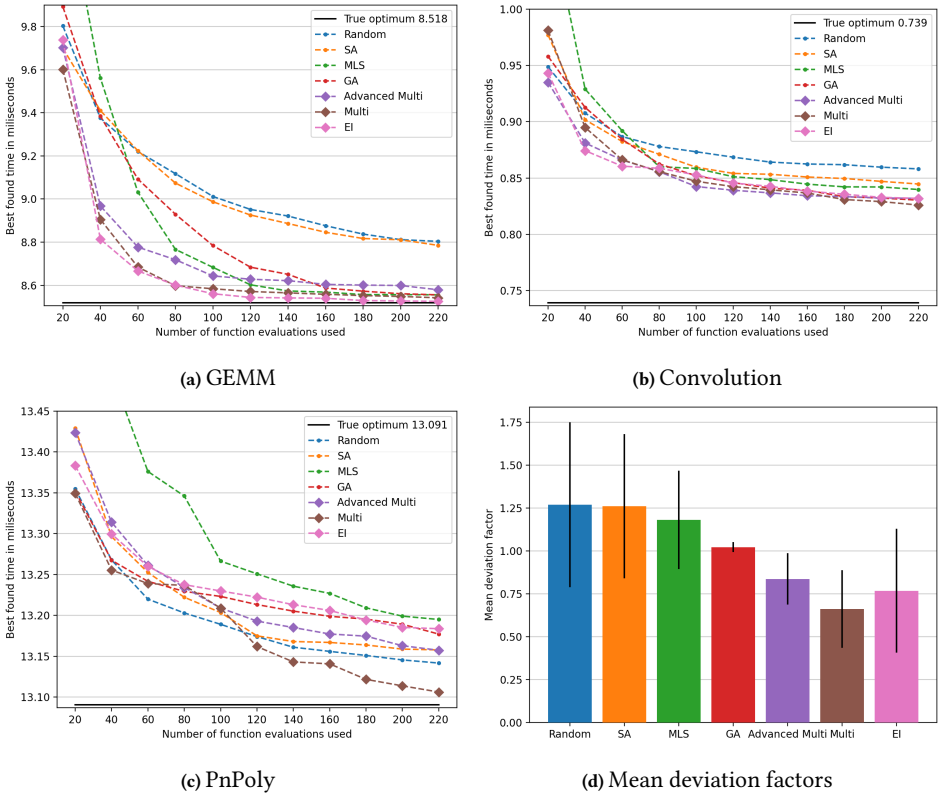
(c) PnPoly **(d)** Mean deviation factors

Figure 2.2: Comparison of search methods on GPU kernels with the RTX 2070 Super; Figs. 2.2a to 2.2c show how the best found performance per method increases as the number of function evaluations increases.

can mostly be explained by the relatively worse performance of *advanced multi* and EI for PnPoly on the A100.

2.4.4 Other frameworks

Following the comparison to the other methods in Kernel Tuner, we now compare our search strategies to alternative BO implementations. These are the widely-used *BayesianOptimization* and *Scikit-optimize* Python packages. We take the defaults recommended by the frameworks, as the hyperparameters used by us are optimized specifically towards our implementation. Neither of these frameworks is able to take search space constraints into account. *BayesianOptimization* uses UCB with exploration factor $\kappa = 2.576$ and 5 restarts for the acquisition function optimizer as the defaults. By default, Scikit-optimize uses *GP-Hedge* with exploration factors $\xi = 0.01$, $\kappa = 1.96$ and 5 restarts for the acquisition function optimizer. We set the number of function evaluations and the number of initial



(a) GEMM (b) Convolution

(c) PnPoly (d) Mean deviation factors

Figure 2.3: Comparison of search methods on GPU kernels with the A100; Figs. 2.3a to 2.3c show how the best found performance per method increases as the number of function evaluations increases.

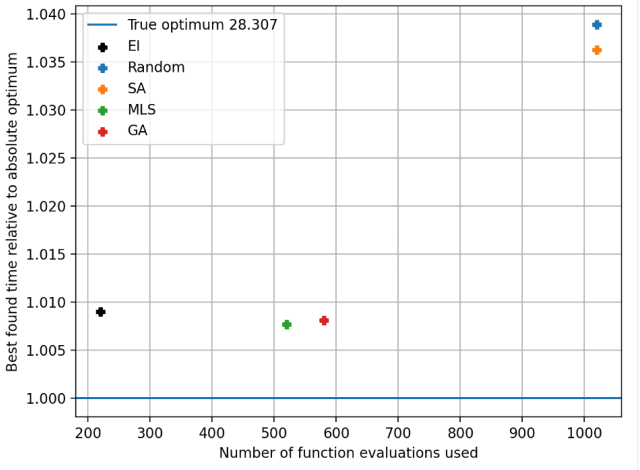
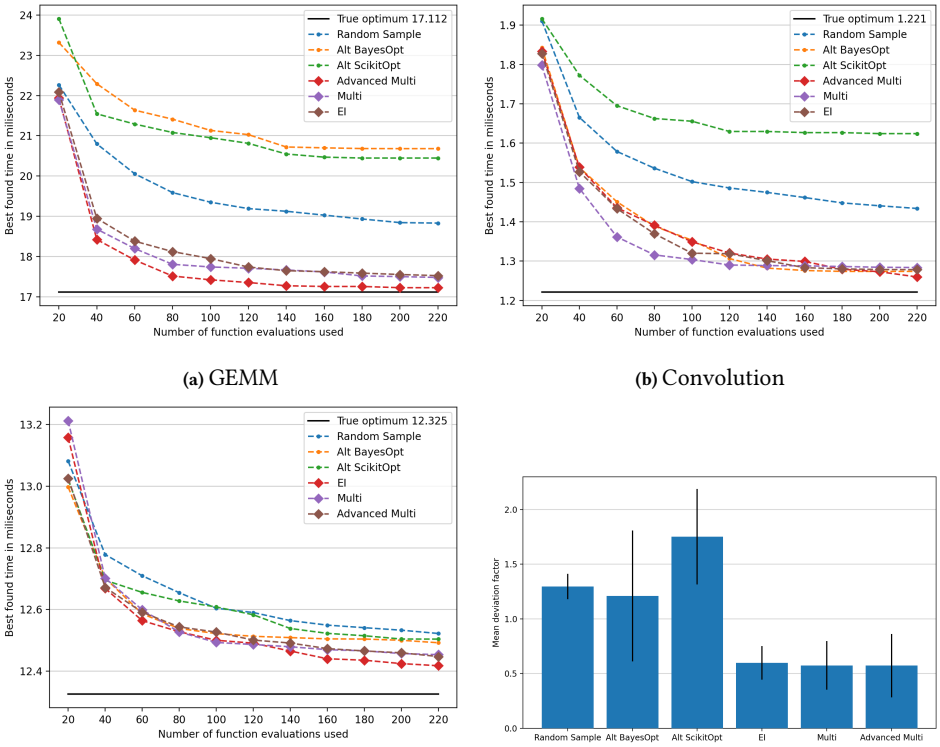


Figure 2.4: Extended relative performance on GEMM with the GTX Titan X



(a) GEMM **(b)** Convolution

(c) PnPoly **(d)** Mean deviation factors

Figure 2.5: Comparison with alternative Bayesian Optimization frameworks for various kernels with the RTX 2070 Super

samples to the same values as before. To create a fair comparison, we use the Nvidia RTX 2070 Super for the comparison among frameworks, as no framework has been optimized for the search spaces on this GPU specifically.

As seen in Fig. 2.5a, the BayesOpt and ScikitOpt implementations perform similar to each other on GEMM, noticeably performing worse than random search. The same holds for ScikitOpt on Convolution. This is likely because for GEMM and Convolution the search space is much smaller with the restrictions applied, which was not possible for these frameworks. However, even in PnPoly, which has no restrictions, performance of ScikitOpt is not much better than random search. *BayesianOptimization* performs better than ScikitOpt on Convolution and PnPoly (Figs. 2.5b and 2.5c). In Convolution, *BayesianOptimization* even performs comparably to our methods. However, as seen in Fig. 2.5d, the general performance of these frameworks does not come close to our BO implementations.

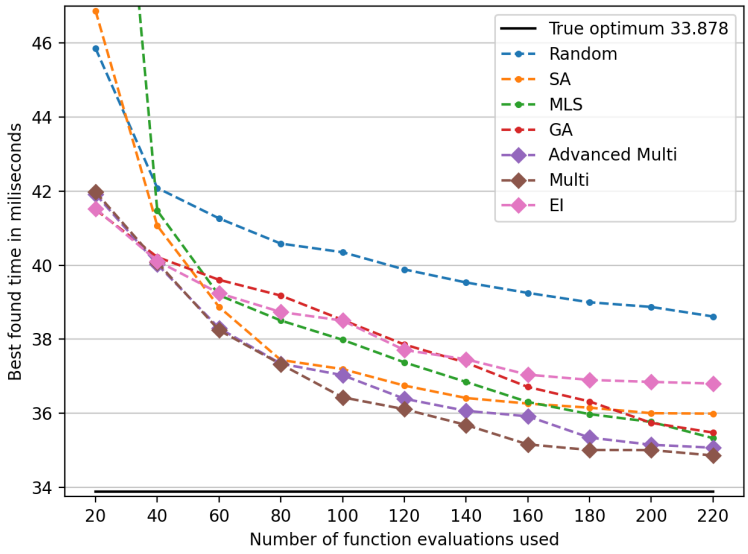


Figure 2.6: Performance on the ExpDist kernel with the A100

2.4.5 Unseen kernels

We have consistently used the three GPU kernels, GEMM, Convolution, and PyPoly, for hyperparameter optimization and for performance comparisons. To verify that our strategies generalize well to other kernels, we also apply it to two previously unseen kernels, namely the ExpDist and the Adding kernels, executed on the NVIDIA A100.

The ExpDist CUDA kernel is a double-precision GPU implementation of a simplified Bhattacharyya distance between two point sets that explicitly takes anisotropic localization uncertainty into account [69]. The tunable parameters describe the thread block dimensions, work per thread in two dimensions, partial loop unrolling factors, and the number of thread blocks to use in the y-dimension. After applying constraints, the search space size consists of 14400 parameter configurations, of which 7320 (50.8%) are invalid. The minimum in time lies at 12.764, however, the work executed by this kernel depends on the parameter configuration, so optimizing on time would result in the kernel that does the least work. Instead, we take $\frac{10^5}{GFLOP/s}$ as an alternative measure, with its minimum at 33.878. Observing Fig. 2.6, *advanced multi* and *multi* perform very well. EI performs worse than the other Kernel Tuner methods and not nearly as well as the *multi* strategies.

The Adding CUDA kernel computes the transport of diffuse radiation through a vertically layered atmosphere [122]. The tunable parameters describe the thread block dimensions in x and y, a partial loop unrolling factor for the second loop in the kernel, which performs 140 iterations, and finally a switch whether or not to store a value computed in

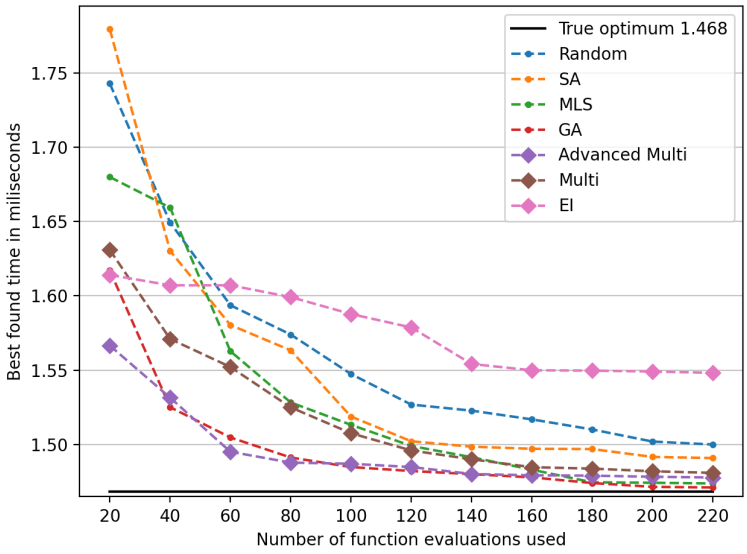


Figure 2.7: Performance on the Adding kernel with the A100

the first loop and load it again in the second loop, or to simply recompute that value in the second loop. The search space after filtering is relatively small, consisting of just 4654 parameter configurations with none invalid, and a minimum of 1.468. In Fig. 2.7 it can be seen that EI appears unable to optimize well, performing worst of all search strategies. MLS, GA, *advanced multi* and *multi* appear to perform similarly to each other.

2.4.6 Discussion

Across all kernels and GPUs, *advanced multi* performed best in general, while the strongest competitor against our BO-methods in Kernel Tuner was a Genetic Algorithm. Looking at the mean deviation factors on the A100 (including the expdist and adding kernels), we find that *advanced multi* performs 19.8% better than GA. Moreover, on the GTX Titan X, *advanced multi* performs 65.6% better than GA, and on the RTX 2070 Super, *advanced multi* performs 63.6% better than GA. Thus on average, *advanced multi* performs 49.7% better at finding parameter configurations than the best other method (GA) in Kernel Tuner. In addition, it performs 75% better than the next-best other method, Simulated Annealing.

2.5 Conclusions

An important aspect of this research has been the constraints of the problem domain, GPU auto-tuning. The challenging aspects are a combination of discrete search spaces,

constraints, rough covariance functions, and invalid configurations, providing a setting in which Bayesian Optimization has not been applied before. To take on these problems, we implemented BO using a mapping to a discrete search space, with a Matérn covariance function, where parameter values are normalized and acquisition functions solely optimized on the non-evaluated parameter configurations. In addition, we propose a Contextual Variance function as an exploration factor to dynamically guide exploration, and propose the *multi* and *advanced multi* acquisition functions to improve scalability and make an informed selection of basic acquisition functions. Finally, we have optimized the hyperparameters of our method to arrive at a set of default parameters that should perform well for most tunable kernels.

To compare the performance of our methods, we used the GEMM, Convolution and PnPoly kernels on three different GPUs, and the ExpDist and Adding kernels on the A100 GPU. Our search strategies *consistently* performed well, outperforming most existing search strategies in Kernel Tuner. Where some existing search strategies perform well on some test cases, they often perform relatively poor on others. On average over the mean deviation factors, *advanced multi* performs 49.7% better at finding parameter configurations than the best other method (GA) in Kernel Tuner, and 75% better than the next-best other Kernel Tuner method (SA). We conclude that in general, *advanced multi* is the best performing method, with a wide margin from other Kernel Tuner methods. Our methods outperform both the other search methods in Kernel Tuner and the alternative BO-frameworks. We conclude that it is not simply Bayesian Optimization in itself, nor the setup provided by Kernel Tuner, but our specific novel implementation that results in such a sizeable improvement over other methods.

3 A Methodology for Comparing Optimization Algorithms

> *Without standards, there can be no improvement.*
>
> attributed to Taiichi Ohno

Auto-tuners rely on optimization algorithms to explore vast search spaces, yet comparisons of such algorithms across studies are hindered by inconsistent experimental setups, budgets, and reporting. Building on input from the 2022 Lorentz Center workshop, we propose a community-driven methodology covering experimental design, tuning budget selection, handling stochasticity, and performance quantification. The approach adapts ideas from related fields while addressing the specific constraints of auto-tuning. We validate the methodology in a case study comparing several algorithms for tuning CUDA kernels on modern GPUs. To promote adoption and reproducibility, we provide an open-source tool implementing the methodology.

This chapter is based on "A methodology for comparing optimization algorithms for auto-tuning" by Willemsen, Schoonhoven, Filipovič, Tørring, van Nieuwpoort, and van Werkhoven [173].

3.1 Introduction

The field of auto-tuning research has concerned itself with developing and applying optimization algorithms to find the optimal parameter configurations as efficiently as possible.

However, while publications on optimization algorithms in auto-tuning often report improvements in these areas, each publication evaluates the performance of optimization algorithms according to their own evaluation methods, metrics, and procedures for benchmarking. The lack of a shared, sound methodology makes comparison and reproduction of results hard.

In March 2022, an international group of researchers in the auto-tuning field gathered in Leiden, the Netherlands for a Lorentz Center workshop titled "Generic Autotuning Technology for GPU Applications", to discuss the state of the field and collaboratively move forward. The lack of a unified methodology to compare optimization algorithms was identified as one of the major challenges towards advancing the field of auto-tuning. The unified methodology we present in this chapter is the result of the collaboration among the different research groups participating in the workshop.

Our community-driven methodology makes the following contributions:

- An overview of the methodological state of auto-tuning research.
- Recommendations for the description of implementation, experimental design, and reporting of optimization algorithm performance comparisons.
- Novel methods for performance comparison:
  - A method to consistently and objectively determine when to stop optimization.
  - A method to approximate the performance of random search to serve as an implementation-independent baseline for comparing optimization algorithm performance.
  - A method to aggregate performance of optimization algorithms across multiple search spaces.
  - Metrics and visualizations focused on reporting the true time spent, instead of the number of function evaluations.
- A jointly developed software package for simple application of the methodology.

The chapter is structured as follows. Section 3.2 shows why a methodology for the auto- tuning field is needed in a threefold approach: the potential distortion of results, methodological limitations in practice, and how similar issues have been addressed in other fields. In Section 3.3, the proposed methodology is explained in detail. Section 3.4

evaluates the methodology by demonstrating it on a use case. Section 3.5 discusses related work focussing on the comparison of optimization algorithm performance, and how other fields have implemented methodologies. Section 3.6 provides concluding thoughts and remarks.

With this methodology, we provide an open source software package that implements the methods presented in this chapter to allow for easy application by other authors.

3.2 State of the Practice

This section demonstrates the need for a novel methodology for the autotuning field by first explaining the potential pitfalls in presenting performance results of optimization algorithms in auto-tuning (Section 3.2.1), followed by an examination of the methodology of various papers presenting optimization algorithms or frameworks for auto-tuning (Section 3.2.2), and several examples of how other fields have implemented methodologies and guidelines (Section 3.2.3).

3.2.1 Six ways to fool the masses when giving performance results

To provide an overview of the ways and magnitude with which results can potentially be distorted, we provide six illustrative examples of comparisons between optimization algorithms or their encompassing auto-tuning frameworks that can "fool the masses" (after [6]). This is based on hypothetical scenarios and is not intended to allude to intentional manipulation observed in practice.

First of all, results can be manipulated by selecting or defining favorable metrics. Suppose we have optimization algorithm A, which utilizes a heuristic, and optimization algorithm B, which first obtains a sample of the search space, builds a surrogate model, and then guides the auto-tuning process by updating and evaluating the surrogate model. If we count the number of function evaluations, we can use this as a measure of the efficiency of both methods. Nevertheless, we should not equate this with the true duration of the tuning process, as algorithm B has the hidden costs of model building and updating.

Secondly, changing the scale of the experiments can influence the results. Consider the optimization algorithms A and B again. As the number of function evaluations does not actually show which optimization algorithm finds well-performing configurations faster, we instead measure the time needed to find such a configuration. Suppose we are tuning a kernel for which algorithm B needs only half the function evaluations required by

algorithm A to find a well-performing configuration. In this case, we can easily choose which algorithm wins the competition. If we take a long-running kernel, the time of surrogate model creation will be negligible, so algorithm B wins (Fig. 3.1b). On the other hand, if we take a short-running kernel, the model time dominates, ensuring algorithm A comes out on top (Fig. 3.1a). Although both scenarios can be sensible, the choice between long or short kernel execution times should be primarily driven by the application and representative input problems.

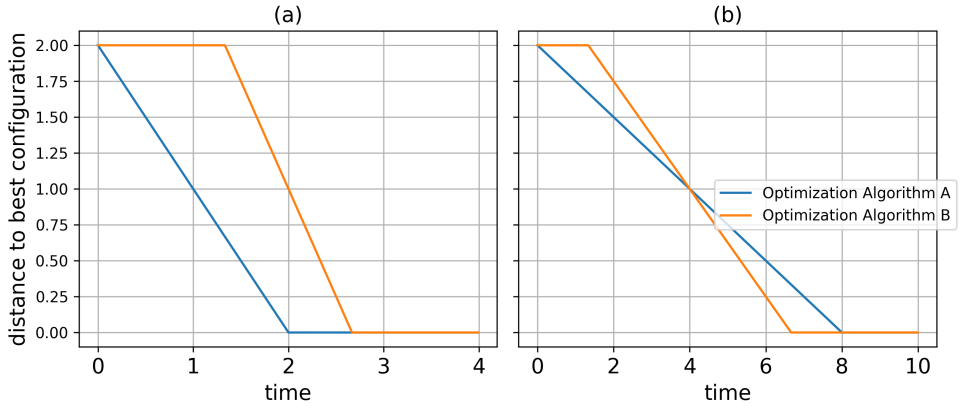


Figure 3.1: Modifying application run-time changes the outcome.

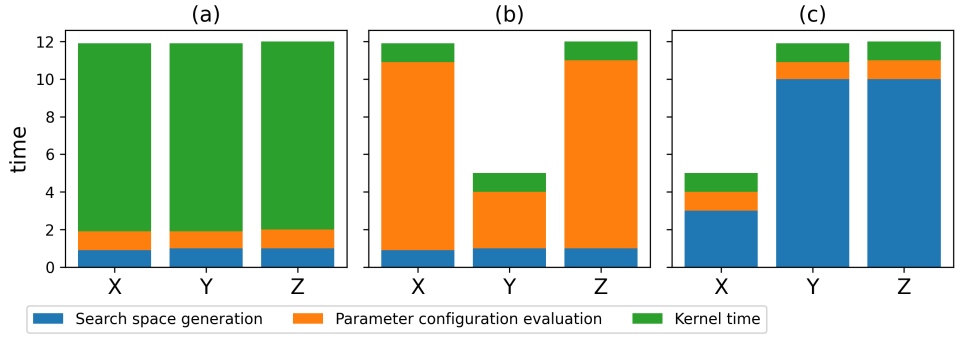


Figure 3.2: Emphasizing differences in overhead changes the outcome.

Performance differences in the implementations of auto-tuners can be exploited by changing the time spent on specific parts of the auto-tuning process. Consider autotuners X, Y,

and Z, each implementing the same optimization algorithm. Auto-tuner X has a computationally efficient method for constructing the search space, which allows it to quickly start the auto-tuning process even if the search space is relatively large. Auto-tuner Y evaluates each configuration slightly faster than X and Z. If the number of function evaluations is counted, auto-tuners X, Y, and Z use an equal number of function evaluations to find a well-performing configuration, as they are using the same optimization algorithm. However, as we are interested in the total tuning time, the results vary depending on the experiment setup. Suppose the authors of Z (which has no advantage over X and Y) want to show that Z matches the performance of X and Y. They thus compare the auto-tuners on a kernel with small or moderately sized tuning space and executing kernels with big inputs, so the overhead of creating the search space and selecting a parameter configuration is masked (Fig. 3.2a). If the authors of Y would like to show their auto-tuner is superior to the others, they set a long tuning-time budget on a small search space, so the high number of evaluated configurations makes their auto-tuner converge in less time than X and Z (Fig. 3.2b). Finally, for X to come out on top, the search space can be extended to make search space generation more demanding and execute it with moderately-sized inputs to suppress faster configuration selection in tuner Y (Fig. 3.2c).

Not just the size of the search space, but also the quality of the configurations in it can influence the results. The overall tuning time is influenced by the number and quality of kernels being benchmarked: if many poorly performing kernels are executed, the searching process is slow. Adding many bad configurations can artificially improve the results of methods that sparsely explore the tuning space and then focus on an area of the well-performing kernels over simpler searchers like random search.

Results can be influenced by their environment as well. The kernel execution time not only depends on kernel settings but also on the environment (e.g. hardware, software, caching, temperature), which can be influenced. For example, using a low-end or older GPU, or even increasing the temperature to cause GPU throttling, increases the time spent in benchmarking configurations, thus favoring optimization algorithms that use fewer function evaluations.

Finally, the choices between implementations of optimization algorithms and frameworks matter. Different auto-tuners can have different implementations of seemingly similar

functionality, such as optimization algorithms and time measurements. For example, as seen in Chapter 2, there are substantial performance differences between different Bayesian Optimization implementations, hence one could take a poorly performing or poorly suited implementation to favorably compare against this type of optimization algorithm in general.

3.2.2 Methodological limitations in literature

In this section, we examine the methodology of various papers presenting optimization algorithms or frameworks for auto-tuning. This is not intended to question or challenge the contributions of these works or authors, but to highlight the frequency of limitations that can influence the interpretation of results.

Ansel, Kamil, Veeramachaneni, Ragan-Kelley, Bosboom, O'Reilly, et al. [3] introduce OpenTuner, a framework for generating domain-specific multi-objective auto-tuners. It uses ensembles of optimization algorithms to gradually select the best-suited optimization algorithms. Most of the optimization algorithms used in the ensemble are only mentioned by name.

CLTune is an auto-tuning framework for OpenCL kernels proposed by Nugteren and Codreanu [114]. The hyperparameters and termination conditions for the optimization algorithms used are not specified. It uses violin plots to concisely portray the distribution of proposed optimization algorithms. Violin plots can be unintuitive, and can be misleading when the distribution cannot be accurately modeled. Without being accompanied by more absolute numbers, it also masks overhead. In addition, each violin plot within the same graph is individually normalized, which gives the initial impression that all optimization algorithms consistently return the optimum, while none actually outperform random search.

Hayes, Li, Chavarría-Miranda, Song, and Zhang [63] introduce ORION, a closed-source framework to tune GPU occupancy, which functions like a single-variable hill-climbing algorithm. This is evaluated using a well-explained benchmark set of twelve kernels on two GPUs.

MATOG is an array layout auto-tuner for CUDA proposed by Weber and Goesele [164]. MATOG abstracts CUDA memory accesses and optimizes the array layouts to the GPU. They evaluate six kernels on eight GPUs, providing a rationale for their hardware sample selection, and using relative speedup to compare performance across GPUs and kernels.

An earlier work of the contributing author van Werkhoven [165] introduces the Kernel Tuner auto-tuning framework, providing several optimization algorithms. These optimization algorithms are evaluated using three kernels on two GPUs. While van Werkhoven

[165] is the first auto-tuning paper to exhibit optimization algorithms that consistently outperform random search, the conclusions do not include a quantified difference between the optimization algorithms and random search.

The Auto-Tuning Framework (ATF) introduced by Rasch, Haidl, and Gorlatch [125] provides improved search space generation in case of constraints on the search space. ATF implements simulated annealing and the Area-Under-Curve bandit technique (which is also part of OpenTuner). They evaluate on one CPU and GPU with a single GEMM kernel using four input sizes. Bar charts show the relative speedup against CLTune and OpenTuner, yet the tuning duration is undefined. The search spaces are different for each framework, which is done to demonstrate the limitations of the other frameworks, but hampers direct comparison. Some of the limitations have been addressed by the authors in a more recent paper [126].

Stachowski, Fiebig, and Rauber [144] use optimization algorithms for tuning frequency scaling to improve the energy efficiency of blockchain algorithms on GPUs, with the goal of switching to the most profitable blockchain algorithm. The experimental setup, software environment, hyperparameters, and stop conditions are not discussed. As just two parameters are tuned, the search space is relatively small.

Bergstra, Pinto, and Cox [19] build a model using boosted regression trees, to offer the speed of model-based auto-tuning with the generality of empirical auto-tuning. This is evaluated on a single kernel. While the hyperparameters and rationale behind the hyperparameters are detailed, a stop condition is not mentioned.

Dastgeer, Li, and Kessler [37] create and evaluate the use of a decision tree for choosing among templates in a skeleton programming library. Detailed descriptions of the benchmarking setup are not included. The hyperparameters and stop condition are not mentioned, and the axes in the figures are not labeled.

Chen and Hollingsworth [29] propose a hierarchical optimization algorithm to Pareto fronts for multi-objective online auto-tuning. This is evaluated on synthetic test cases. The Pareto front approximation is demonstrated on a relatively small search space of 320 configurations for a single GPU. The source code is not available online.

Guerreiro, Ilic, Roma, and Tomás [60] focused on optimizing a concurrently executed set of kernels, where parameters influence all of the kernels. They propose two algorithms, for execution time and energy optimization respectively, and a multi-kernel version for each. The optimization algorithm runs for at most 8 function evaluations.

Czarnul and Rościszewski [35] present an expert-knowledge-based heuristic algorithm for minimization of parallel application execution times on modern hybrid CPU+GPU systems. The involvement of expert knowledge is not discussed in detail, nor is an analysis

of the sensitivity to the quality of the expert knowledge included. The hyperparameters and stop condition are not mentioned, nor is the source code of the algorithm available online.

Wu, Kruse, Balaprakash, Finkel, Hovland, Taylor, et al. [176] adds Bayesian Optimization to the loop optimization pragma auto-tuning framework YTOPT. This is evaluated on a subset of the PolyBench benchmark suite. The hyperparameters and stop conditions are not mentioned. The presented results are single runs, while the optimization algorithm is stochastic.

The earlier Chapter 2 introduced Bayesian Optimization to Kernel Tuner [165]. This is evaluated against the other optimization algorithms in Kernel Tuner using three kernels on three GPUs. Because of the large number of compared algorithms, some error values are omitted. Although the conclusion is quantified, this is done with the self-defined "mean deviation factor" metric.

Li and Agrawal [89] propose an optimization algorithm called *shrinking sample*, which takes a hierarchical approach starting from a global view and iteratively shrinks the search space until a final subset of the space is searched exhaustively. They compare the performance of their method to several optimization algorithms present in Kernel Tuner 0.2.0 [165]. For each method, the hyperparameters are tuned and the performance of the best-performing configuration is used in the paper. The results are shown for a single GPU. The importance of individual tunable parameters is analyzed for several of the benchmark applications. The exact values for the hyperparameters of the optimization algorithm are not included in the paper, nor is the source code of their algorithm available online.

In earlier work, the contributing authors Schoonhoven, van Werkhoven, and Batenburg [134] conducted a survey of auto-tuning optimization by performing experiments for 26 different search spaces on 9 different GPUs using 16 different evolutionary black-box optimization algorithms. They introduce a PageRank centrality metric to quantify the search space difficulty. Tournament selection is used to rank the optimization algorithms in a comparison. They combine the application of tournament selection with statistical tests. This results in a quantifiable ranking where the magnitude of differences is unknown.

Earlier work of contributing author Filipovič, Hozzová, Nezarat, Ol'ha, and Petrovič [48] introduces hardware performance counters to speed up auto-tuning in Kernel Tuning Toolkit (KTT). It builds a model of relations between tuning parameters and performance counters to navigate the auto-tuning search. Their results display both function evaluations and tuning time in seconds. It evaluates against random search and the Basin Hop-

Category	Limitation	Articles	Percentage
Experimental Setup	Stop condition not described	9	53%
	Single device	4	24%
	Single kernel	4	24%
	Small search spaces	3	18%
Reproducibility	Environment underspecified	10	59%
	Optimization algorithms underspecified	11	65%
	Not open source	7	41%
Reporting	Missing time spent tuning	12	71%
	Relies on speedup	4	24%
	Missing information in plots	7	41%

Table 3.1: Overview of common methodological limitations in a sample of 17 auto-tuning publications.

ping optimization algorithm in Kernel Tuner [165]. The improvement over random search and basin hopping is not quantified.

The 17 publications discussed in this section demonstrate the methodologies used in practice. An overview of commonly encountered methodological limitations is given in Table 3.1, which shows that a large portion of the included publications exhibit various limitations. In some cases, positive examples are noteworthy, such as [164], which extensively evaluates six kernels on eight GPUs and provides a rationale for this selection. However, in general, conclusions drawn from results are not quantified, there is a lack of proper baseline when reporting speedups, and hard-to-interpret metrics and graphs are used. While not intentional, this may hamper the progress of the field as a whole, as it is difficult to get an accurate overview of the state of the art and to draw conclusions by comparing publications.

3.2.3 Related fields

This subsection describes how other fields have identified limitations and implemented methodologies to enable comparisons and increase research quality.

In 1979, Crowder, Dembo, and Mulvey [34] established guidelines for computational experiments in general and mathematical software specifically, providing a checklist of 42 recommended and optional items, detailing what information should be included in papers. They recommend using synthetic tests over tests on real-world data, as the parameters for synthetic tests can be easily shared in the paper.

Since then, several guidelines and methodologies have been published in related fields that observed similar problems. Kitchenham, Pfleeger, Pickard, Jones, Hoaglin, Emam, et al. [80] propose to improve the quality of empirical research in software engineering by introducing a total of 32 guidelines divided into six areas: experimental context, exper-

imental design, conduct of the experiment and data collection, analysis, presentation of results, and interpretation of results. Similarly, Davis [38] proposes guidelines regarding benchmarking, visualization, and statistical analysis in the real-time scheduling domain. In the domain of cloud computing, Papadopoulos, Versluis, Bauer, Herbst, Kistowski, Ali-Eldin, et al. [117] proposes methodological principles for reproducible performance evaluation. They propose eight methodological principles combining best practices from related fields with concepts specific to cloud computing and apply these principles in a use-case experiment.

Bailey [6] humorously raises issues with the manner in which parallel computing results were reported as early as 1991, which was later expanded on by proposing 9 guidelines to avoid these issues [7]. Based on what has been observed in Sections 3.2.1 and 3.2.2, the following guidelines suggested in [7] are relevant. Guideline 1 states that when a benchmark is used, the rules of that benchmark must be followed so the results are comparable to others using the same benchmark. Guideline 4 states that timings should be true elapsed time-of-day instead of derived metrics or rates whenever possible. Guideline 9 states that the details of the hardware and software environment, as well as other details necessary for comparison and reproduction, such as bases for FLOP counts and speedup rates, should be included in papers.

Hoefler and Belli [71] demonstrates limitations in contemporary literature and gives guidelines for reporting results in high-performance computing (HPC). They assess the state of methodology in HPC by evaluating the experimental design and data analysis of 120 papers in three top conferences, proposing 12 guidelines on benchmarks, measurement distribution, documentation of the environment and experimental setup, and visualization of results to improve HPC publications. To facilitate the adoption of their suggestions, they provide their methodology in a C-library named LibSciBench.

Several publications have proposed guidelines and methodologies for specific practices, languages, and applications in HPC. These publications tend to be specific to their practice, but some propose guidelines that are also relevant to our practice. For example, Georges, Buytaert, and Eeckhout [56] propose statistical methods for performance evaluation in Java, and Horký, Libič, Steinhauser, and Tůma [74] propose practical guidelines for performance measurements in Java.

Others attempt novel approaches, such as Collective Mind (also referred to as cTuning) proposed by Fursin, Miceli, Lokhmotov, Gerndt, Baboulin, Malony, et al. [53], which intended to collect auto-tuning data via crowd-sourcing to draw general lessons from auto-tuning. This was discontinued when the author shifted focus towards research artifacts for machine learning [52].

One of the most notable instances of a collaborative effort for methodology and benchmark is MLPerf, which aims to fairly compare machine-learning performance across a diverse set of systems [128]. Driven by a coalition of over 30 organizations and fueled by a collective of more than 200 engineers and practitioners, MLPerf establishes a framework of regulations and preferred methodologies. Similar to [7], the recommended metric *time-to-convergence* is as close as possible to real-world performance as perceived by users [102].

The papers discussed in this subsection illustrate the need for guidelines in other fields and show recurrence in the topics addressed by guidelines: experimental design, baselines, metrics, statistical evaluation, and results presentation and interpretation.

3.3 Methodology Description

The main goal of this methodology is to provide a structured approach for comparisons between optimization algorithms and their implementations for auto-tuning applications.

To provide a high-level summary before detailing the methodology further, the main aspects and actions required are summarized step-by-step:

Step 1 Experimental Setup describes how to decide what tunable applications to use, to make sure search spaces are realistic, and the environment is well-defined, in order to ensure the experiments are representative. Applications are chosen from a representative sample either by analyzing common applications in a domain or using established benchmark suites. The tunable parameters and values are defined and constrained by their significant impact on the objective. The environment selection aims at diversity to reflect different hardware capabilities and ensure a representative testbed.

Step 2 Optimization Algorithms entails detailed descriptions of the algorithms and their dependencies, as well as the hyperparameters and selected values. Care should be taken to justify the selection of implementations of algorithms, and hyperparameters must be consistently applied across different tests without unfair adjustment based on the application or similar mechanisms

Step 3 Tuning Budget and Noise involves two important decisions: how long to auto-tune for, and how often to repeat to obtain reliable results. The methodology combines time-based and value-based budgets using random search to establish a standard for comparison. This step also addresses the handling of noise in the measurements, requiring runs to be repeated often enough to ensure the reliability of the results.

Step 4 Reporting Optimization Algorithm Performance
describes how to obtain metrics and visualizations, with a focus on comparing algorithms across search spaces. To provide an implementation-independent baseline, a calculated random search is introduced. To move from number of function evaluations to real time spent, isotonic regression is applied to get a monotonic curve along a discrete time range. Time spent on a single configuration can be broken down into several categories for further analysis. To compare across search spaces, the baseline can be normalized using the previously established tuning budget. In addition, guidelines for dealing with algorithms that exhibit early-stopping behavior and visualization of the relative performance of optimization algorithms across search spaces over time are provided. Performance is shown as the fraction of the global optimum relative to the random search baseline. Uncertainty is visualized with specified confidence and prediction intervals. These steps lead to a concise, intuitive visualization and quantification of the performance across search spaces.

These concepts and guidelines are elaborated on in Sections 3.3.1 to 3.3.4. From here on, we make several assumptions for simplicity, without loss of generality. We assume the auto-tuning of individual GPU kernels for minimal run time using optimization algorithms. Optimization algorithms are assumed to be single-objective, sequentially processed, and stochastic. The search spaces are discrete and can be pruned by constraints. It is important to note that the methodology can be applied regardless of the optimization objective, even though performance will be used as the objective throughout the examples. The structured approach outlined ensures that each aspect of the optimization algorithm comparison is handled with a clear focus on generating reliable, comparable, and meaningful results.

3.3.1 Step 1: Experimental Setup

Benchmarking optimization algorithms for auto-tuning in a fair, reproducible, and comparable way is difficult. This section explains how to select tunable applications (Section 3.3.1), how to define the tunable parameters (Section 3.3.1), and how to select the hardware and software environment for the experiments (Section 3.3.1).

A representative sample of applications can be selected either by analyzing the population within a scientific domain (e.g. climate modeling, radio astronomy), or by using a selection of common HPC applications (as done by the Parboil [149] and Rodinia [27] benchmark suites, containing e.g. matrix multiplication, hotspot, and stencil applications). For the latter, it is left to benchmark suites to supply an auto-tunable, balanced set of kernels. It is

important that benchmarks provide sufficiently large problems to avoid under-utilization of the hardware in the sample. As per Bailey [7], when using a well-known benchmark suite, it should be left as-is to keep the results comparable. At the 2022 Lorentz Center workshop on "Generic Autotuning Technology for GPU Applications", where the foundations of this methodology were discussed, the need for a benchmark suite for auto-tuning was also addressed. This resulted in *BAT*, the Benchmarking suite for Auto-Tuners[1]. The accompanying paper by Tørring, van Werkhoven, Petrovč, Willemsen, Filipovič, and Elster [159] details how the search spaces are of high quality in accordance with five characteristics: convergence rate, local minima centrality, optimal speedup, Permutation Feature Importance (PFI), and performance portability.

The search spaces are the Cartesian product of the tunable parameters and their values, after application of constraints. The tunable parameter values, and by extension the search space, may seem to follow directly from the implementation of an application. However, the decisions regarding search spaces can influence the performance of optimization algorithms substantially. We thus propose the following guidelines:

- The tunable parameters, their values, and constraints used should be noted or referred to.
- The search spaces of well-known benchmark suites should be used as-is.
- Only tunable parameters that significantly affect performance, and parameter values that have a reasonable chance of achieving good performance should be included. This prevents artificially increasing the search space construction time or the effectiveness of optimization algorithms. For example, the Permutation Feature Importance (PFI) metric of Tørring, van Werkhoven, Petrovč, Willemsen, Filipovič, and Elster [159] can be used to restrict the search space to parameters that have importance in the outcome.
- Conversely, search spaces should not be unreasonably small, for example by prematurely optimizing the search space to a subset such as the current hardware configuration. The Proportion of Centrality metric of Schoonhoven, van Werkhoven, and Batenburg [134] can be used to quantify the difficulty of a search space.
- Finally, it is important to note that search space quality may differ between optimization objectives, e.g., a search space designed for optimizing time may not be of the

[1] https://github.com/NTNU-HPC-Lab/BAT

desired quality when optimizing for energy efficiency. Therefore, we recommend to apply the above steps for each optimization objective separately.

To a large extent, the selected applications drive the selection of hardware and software. We recommend to aim for a representative testbed, such as selecting hardware from different vendors, different grades, and different time periods. For example, for CUDA GPUs, the Compute Capability (*CC*) version indicates the feature set available in hardware and software, and can thus be used to decide which micro-architectures are relevant. Major differences are indicated by micro-architecture, which allows for creating a representative sample based on the micro-architectures instead of the individual GPUs, as done by Weber and Goesele [164] and Filipovič, Hozzová, Nezarat, Ol'ha, and Petrovič [48].

To ensure reproducibility of the experiments, all relevant hardware and software information should be recorded, including operation system and driver versions. This can be achieved by providing a *software bill of materials* that allows the software environment to be rebuilt. Software environment files can be provided depending on the implementation (e.g. Conda, NPM, Docker, Singularity).

3.3.2 Step 2: Optimization Algorithms

All evaluated optimization algorithms (including those compared against) should be described in sufficient detail to reproduce the results or have their source code publicly available. Care should be taken that dependencies of optimization algorithms are part of the software specification as well.

Differences between implementations of an optimization algorithm can have a strong influence on performance (for example seen in Chapter 2). To avoid exclusively evaluating against optimization algorithms that perform poorly while more suitable implementations are available, it is necessary to provide justification for how the set of optimization algorithms has been selected.

Hyperparameters influence optimization algorithm performance, thus specification of the hyperparameters and their values is needed. The assumptions and circumstances of the hyperparameter tuning process should be described in detail.

Hyperparameters are usually constants, but sometimes a more dynamic approach is applied. In this case, the effects of the formulas should be discussed to ensure no prior knowledge of the benchmarks is inadvertently included. In addition, the hyperparameters should be consistent, that is, to not change based on the evaluated application or related factors if this yields an unfair advantage. Given a representative sample of the problems and environment within the scope as per Section 3.3.1, hyperparameter tuning on the

entire set results in balanced hyperparameters, but changing these hyperparameters based on individual problems or environment factors results can result in an unfair advantage as it is no longer representative of the performance for an untrained problem.

3.3.3 Step 3: Tuning Budget and Noise

To ensure a fair comparison among different algorithms, we need an objective way to determine how long to tune for (Section 3.3.3) and how often to repeat the experiment to minimize noise (Section 3.3.3).

The experimental or auto-tuning budget determines how long optimization algorithms are at most allowed to run. This duration is commonly defined in the number of function evaluations in the field of optimization algorithms, but as in Bailey [7] and Mattson, Reddi, Cheng, Coleman, Diamos, Kanter, et al. [102], wall-clock time passed is the more relevant metric for auto-tuning. Ideally, this time is as short as possible without missing the best configurations.

In general, there are two kinds of stop conditions: either meeting a target objective value (*fixed-target*) or reaching a maximum budget spent (*fixed-budget*). The objective value can be an absolute value in the search space, or a value relative to this (e.g. 95% of the optimum). The budget can have any arbitrary definition, but in general, this is a number of function evaluations, an amount of time, or some other limiting factor such as energy consumption or cloud credits. It is important to note that the budget is not necessarily directly related to the optimization objective, for example when optimizing a program for energy consumption while having a time budget.

Both types of conditions are hard to compare across search spaces when using fixed values. For example, the meaning of ”95% of the optimum” or ”200 function evaluations” may be quite different between two search spaces and can thus influence results, as shown in Section 3.2.1. Instead of choosing between the two types of conditions, we propose to combine both using random search, as random search incorporates the characteristics of the search space, providing a baseline that optimization algorithms should aim to beat. When using the number of function evaluations, we propose setting the budget to the number of function evaluations taken by random search to reach a target fraction of the distance between the global optimum and median objective value. The median is used to be more robust against outliers in the search space.

When comparing performance over wall-clock time, the budget in time can be approximated by multiplying it by the average time per function evaluation. Because invalid

configurations (those that failed to run, for example, because they resulted in a compilation error) do not have an objective value that can count towards the results, the average time per function evaluation must include the average time of invalid configurations. This can be calculated by multiplying the median time per invalid function evaluation with the fraction of invalid function evaluations in the search space, and adding this to the median time per valid function evaluations, resulting in the mean time per function evaluation m_f. In essence, other optimization algorithms are given the number of function evaluations or equivalent amount of time it takes random search to reliably reach values close to the optimum.

This definition of the tuning budget has the advantage that it is invariant across autotuning frameworks, optimization algorithms, and tunable applications, as it can be calculated directly, as long as the full search space has been evaluated. To calculate the tuning budget, first find the objective value at the cutoff point with

$$c_o = g + ((m - g)(1 - p)) \tag{3.1}$$

where g is the global optimum objective value, m is the median objective value, and p is the target fraction. The latter determines at which distance between the median and optimal objective value the cutoff point is. Next, get the index c_i of the first element where $o \leq c_o$ (assuming minimization) of the descending sorted list of all objective values $o \in O$ in the search space. Now the number of function evaluations needed by random search to reliably find a configuration at least as good as the cuttoff c_f can be calculated as:

$$c_f = \left\lceil \frac{c_i}{|O| + 1 - c_i} \right\rceil \tag{3.2}$$

In section Section 3.3.4 we revisit random search and describe how the c_i in Eq. (3.2) can be determined. Finally, the approximate cutoff in wall-clock time c_t can be calculated by

$$c_t = c_f * (\widehat{T_v} + \widehat{T_i} \cdot (\frac{|T_i|}{|T_v \cup T_i|})) \tag{3.3}$$

where T_v is the time for each valid configuration, T_i is the time for each invalid configuration, and $\hat{x}$ denotes the median of x.

To ensure valid results when aggregating across search spaces, it is important to use a consistent target fraction p for all experiments. As a rule of thumb, we propose to use $p = 0.975$. This gives the time taken by random search to reach a parameter configuration within 2.5% from the difference between the search space median and the global optimum, as beyond this point where random search reliably finds values close to the global op-

timum, there is arguably little benefit to using an optimization algorithm over random search.

Then there is the second decision: how often the auto-tuning process should be repeated for reliable results, which essentially consists of two stochastic elements: noise from the measurements themselves, and stochasticity in the optimization algorithms. This noise can be reduced by repeating the underlying process. We must thus determine the number of times the measurements and experiments should be repeated to obtain reliable results.

For individual measurements (that is, the evaluation of a single parameter configuration), the Central Limit Theorem [177] can be applied, as these are individual and identically distributed random values. This is illustrated by taking a random single parameter configuration of the GEMM kernel of Chapter 2 on the RTX 2070 Super, and plotting the distribution of its mean as we increase the sample size. As seen in Fig. 3.3, the distribution of the sample mean gets closer to a normal distribution as the number of samples increases. As a rule of thumb, there should be ≥ 30 samples, but this should be based on observations of the stability of the measurements.

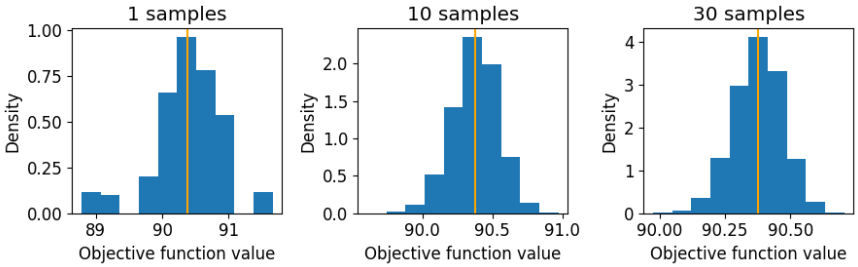


Figure 3.3: Distribution of GPU kernel execution times with three sample sizes in a density histogram, 10000 repeats, orange line is the population mean.

In practice, there are additional considerations regarding the number of samples. For instance, there are other ways in which the noise can be reduced, such as fixing the clock frequency or disabling frequency boost features. In addition, the assumption is that samples from the same configuration are independent and identically distributed, while this may not always be the case. For example, if a rise in GPU temperature causes throttling, this may affect the reported performance, both within a configuration and between configurations. When exploring a search space by brute force, this could be mitigated by iterating over both the configurations and their samples in random order, instead of consecutively. However, this is not the case with optimization algorithms, and would thus

not be representative of the performance to be expected. Instead, additional research is required to look into this problem.

The stochasticity of optimization algorithms calls for a different approach, given that the optimization algorithms are represented by the best-found-value-so-far, i.e., monotonic non-increasing for minimization and non-decreasing for maximization functions. The Central Limit Theorem does not apply here, as the results of optimization algorithms cannot be assumed to be individually and identically distributed random values, and for optimization algorithms that take a relatively long time to run it may not be feasible to obtain a large sample. Therefore, we propose using non-parametric confidence and prediction intervals to visualize the variance (further detailed in Section 3.3.4), and hence to visually confirm whether the number of times an optimization algorithm has been executed is sufficient.

3.3.4 Step 4: Reporting Optimization Algorithm Performance

To go from results on experiments to valid comparisons, this section details how the results obtained via the previous stages can be processed and presented as quantities and visualizations. First, to enable general comparisons, Section 3.3.4 proposes a method for a calculated baseline. Following this, we look at how an individual search space over function evaluations and time can be presented in Section 3.3.4 respectively, and how the elapsed time can be more closely examined in Section 3.3.4. Next, comparisons across search spaces are detailed in Section 3.3.4. Section 3.3.4 validates the baseline, and Section 3.3.4 provides instructions on dealing with early-stopping algorithms. Ultimately, Section 3.3.4 details how the performance of an optimization algorithm can be quantified in a single number.

To enable general comparisons across search spaces, a baseline is needed. Given the large variation in search spaces [134], such a baseline needs to take the characteristics of search spaces into account in order to be stable.

There are several ways in which aggregating results across search spaces has been approached in literature (see Section 3.5). The performance profiles introduced by Dolan and Moré [43] are empirical cumulative distribution functions (*ECDFs*) relative to the best performing optimization algorithm per benchmark, and similarly, Hansen, Auger, Ros, Mersmann, Tušar, and Brockhoff [61] uses absolute ECDFs. However, this uses either a fixed-budget or a fixed-target, and the dependence on a selected ”best performing” optimization algorithm introduces additional problems, such as the ranking problem described by Gould and Scott [59].

Instead, we propose to use random search as a baseline (similar to the tuning budget of Section 3.3.3), as random search provides an intuitive lower bound on expected performance regardless of the search space, is transparent, and is largely implementation independent. Whereas the tuning budget formula in Section 3.3.3 calculates the number of function evaluations random search is expected to need to reach an objective value, the random search baseline works inversely: given a number of function evaluations, it provides the expected objective value reached by random search.

This expected objective value can be calculated with a closed-form expression that can be derived as follows. A search space O can be considered as an ordered list $X = x_1 < x_2 < \ldots < x_N$ of objective values (where $N = |O|$). Random search draws from this list without replacement and keeps the best value found. Mathematically, this is the same as drawing indices uniformly at random without replacement from $1, \ldots, N$ and keeping the highest index drawn.

Suppose we draw M times, and draw the values $y_1 < y_2 < \ldots < y_M$. The probability that y_M is exactly x_M (the lowest value we could possibly draw in M attempts) is very small, namely, $P(y_M = x_M) = \frac{1}{\binom{N}{M}}$ since there are $\binom{N}{M}$ ways to draw M numbers from N, and exactly one way where $y_M = x_M$. In general, the probability that y_M is equal to x_{M+i} is

$$P(y_M = x_{M+i}) = \frac{\binom{M+(i-1)}{M-1}}{\binom{N}{M}}.$$

This equation gives us a probability that the best index we draw happens to be $M+i$. Next, the expected value of a quantity is calculated by multiplying the quantity by its probability, and summing the result; $\mathbb{E}[Z] = \sum_i z_i P(Z = z_i)$. Therefore, the expected best index to be drawn after M draws is

$$\begin{aligned}
\mathbb{E}[\text{Index of } y_M \text{ in X}] &= \sum_{i=0}^{N} i \cdot P(y_M = x_i) && (3.4)\\
&= \sum_{i=0}^{N-M} (M+i) \cdot P(y_M = x_{M+i}) && (3.5)\\
&= \frac{1}{\binom{N}{M}} \sum_{i=0}^{N-M} (M+i) \binom{M+(i-1)}{M-1} && (3.6)\\
&= \frac{M(N+1)}{M+1}. && (3.7)
\end{aligned}$$

In the first step we use $P(y_M = x_i) = 0$ for $i < M$ since we draw M times. If we remember that M is the number of function evaluations f, and $N = |O|$ is the size of the search space,

Eq. (3.8) gives the expression for the expected index of the best value that random search will find in f evaluations:

$$r_i = \left\lfloor \frac{f \cdot (|O|+1)}{f+1} \right\rceil. \tag{3.8}$$

To get the random search baseline, we simply calculate r_i for every f in the range F, and lookup the value O_{r_i}. This yields a monotonic step-wise curve. To interpolate this curve to a smooth curve, the monotonicity-preserving Piecewise Cubic Hermite Interpolating Polynomial of [51] is applied. Note that the cut-off number of function evaluations in Eq. (3.2) can be determined by using $r_i = c_i$ in Eq. (3.8), and solving for f.

The baseline can be applied by subtracting it from the curve of each optimization algorithm (demonstrated in the bottom plots of Figs. 3.4 and 3.5). This yields for every point in the time range of interest a performance score where below 0 indicates worse than random, above 0 indicates better than random, with a maximum of 1 (the global optimum has been found).

When comparing optimization algorithms on a single search space, it is trivial to do so using the number of function evaluations. As a given range of function evaluations from the start to the cutoff point is discrete, performance can be directly compared at each number of function evaluations in the range. A function evaluation is counted for any evaluated configuration, even if it fails to produce a valid result (e.g. compilation or runtime failure), unless a configuration has been previously evaluated or does not meet the search space constraints (i.e. it is assumed frameworks cache previous evaluations of configurations and assess only configurations that meet the constraints).

Visualizing the performance of an optimization algorithm over the number of function evaluations is trivial: at each function evaluation, take the average best-found function evaluation. General consensus for visualizations over time is to have the time domain on the x-axis, leaving the performance metric on the y-axis.

With the number of function evaluations on the x-axis, the uncertainty in the data can be visualized by means of a confidence interval at each number of function evaluations. For a confidence level of 95%, the confidence interval gives a range of values that contain the true value with a 95% probability. Non-parametric confidence intervals can give these intervals even if the underlying distribution is unknown [71]. As per Le Boudec [86], for large n the approximation of Eq. (3.9) can be used for the lower rank and upper rank respectively, where $v = \eta\sqrt{np(1-p)}$. Here η is the lookup of probability $\frac{1+\gamma}{2}$ on the

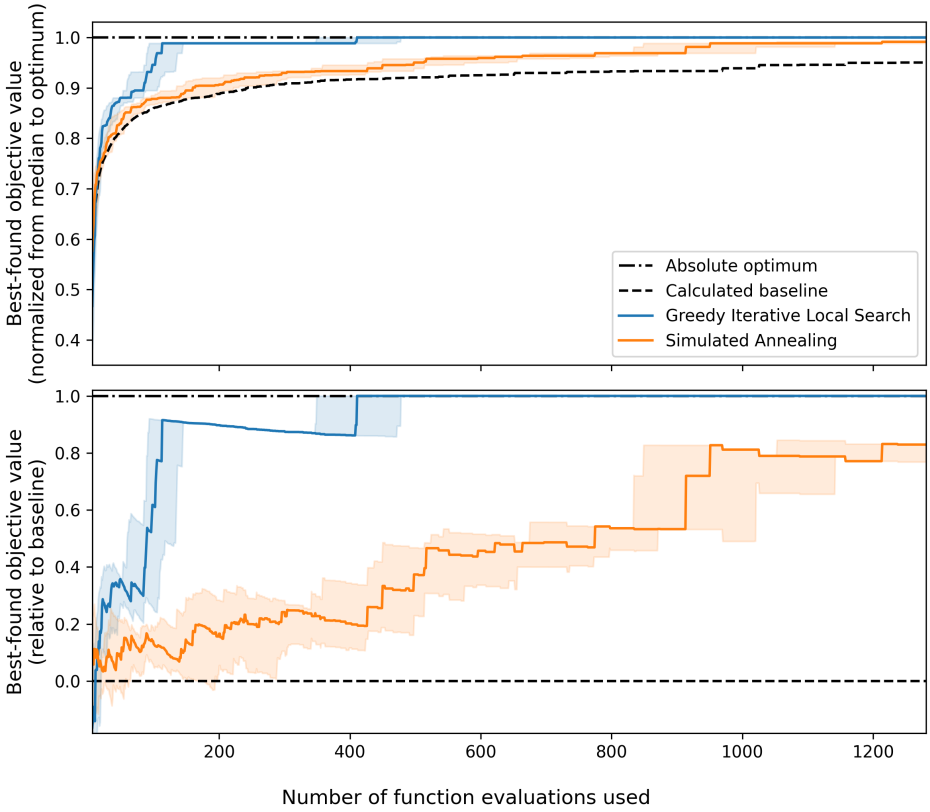


Figure 3.4: Example visualization of optimization algorithm performance over function evaluations on a single search space.

inverse cumulative distribution function on a normal distribution, with confidence level γ (e.g. for $\gamma = 0.95$, $\eta \approx 1.96$).

$$r_l, r_u = \lfloor np - v \rfloor, \lceil np + v \rceil + 1 \tag{3.9}$$

An example of the visualization of an individual search space over function evaluations is shown in Fig. 3.4. The top plot shows the best-found values by each algorithm after a number of function evaluations, normalized from the search space median to the optimum. The bottom plot shows the same values but relative to the baseline.

Though both plots show the same data, they serve a different purpose. In the bottom plot, the random search baseline and global optimum are horizontal lines at the y-values 0 and 1 respectively, opaquing the fact that the absolute distance between the baseline and global optimum decreases as the calculated baseline finds increasingly better options. For this reason, it is accompanied by the top plot showing absolute or normalized performance

values, which will show asymptotic curves converging to the global optimum. Hence the top plot is geared towards an intuitive, global understanding of the data, while the bottom plot provides more details. For example, in the bottom plot the optimization algorithm "Greedy Iterative Local Search" performance seemingly declines from function evaluation 150 to 400, but looking at the top plot reveals that this is simply because the baseline improves while the optimization algorithm is stagnant. In both plots, the shaded areas mark the confidence interval.

As mentioned previously, the number of function evaluations is not necessarily an accurate indicator of the performance over time for auto-tuning. Instead, the wall clock time is more indicative of expected performance. Nevertheless, practically no papers in our practice report time in this way (notable exceptions being Petrovič, Střelák, Hozzová, Ol'ha, Trembecký, Benkner, et al. [121] and Filipovič, Hozzová, Nezarat, Ol'ha, and Petrovič [48]). With good reason: time is not as trivial as using the number of function evaluations because it is a continuous domain, so unlike function evaluations, performances over time cannot be compared directly.

However, we know that each individual run is monotonically non-increasing assuming minimization, or monotonically non-decreasing assuming maximization, and as such, an aggregate of an optimization algorithm should adhere to this as well. Isotonic regression is a technique for fitting observations to a free-form curve while maintaining monotonicity [13]. Using isotonic regression, we can create a piecewise monotonic curve along a discrete, linear time range that ends at the cutoff point of Section 3.3.3. As this time range is discrete, it needs a consistent resolution. To avoid presenting the data as smoother than it is in reality and mitigate overfitting [100], we recommend selecting a resolution within the order of magnitude of the smallest average total evaluated configurations within the budget for each algorithm. For example, if in one search space algorithm A has an average of 1000 total evaluated configurations, and algorithm B has an average of 100, the overall resolution should be in the order of the latter. This also incentivizes increasing the number of times an optimization algorithm is repeated. Accounting for the ratio of early stopping (discussed later) is allowed. Using isotonic regression in this way, the optimization algorithms are directly comparable at every point along this range.

When isotonic regression is used, there is not only uncertainty in the values as there is with the number of function evaluations; there is also uncertainty in the predictions. By instantiating multiple isotonic regressors given different parts of the data (a "*bagging*" approach), this uncertainty can be quantified. We propose to split the valid data along the square root of the number of runs per optimization algorithm r, meaning each instance

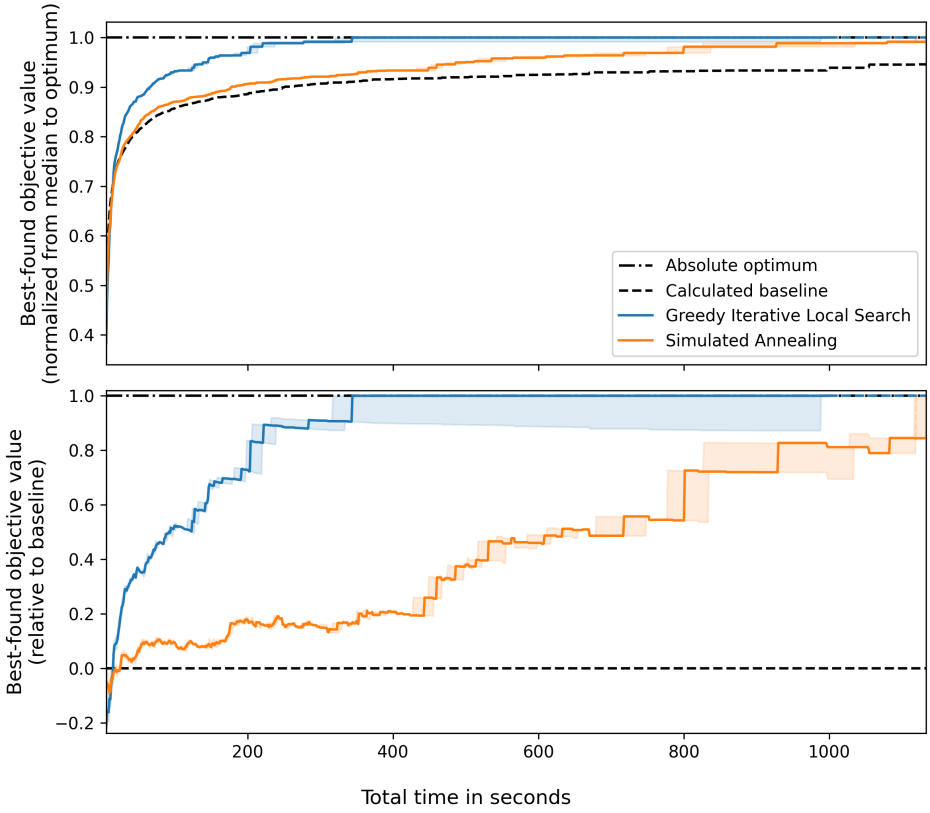


Figure 3.5: Example visualization of optimization algorithm performance over time on a single search space.

only gets a fraction $\frac{1}{\sqrt{r}}$ of the data. This way, data reuse is limited and differences in predictions reflect the variance in the data.

The baseline of Eq. (3.8) uses a number of function evaluations f as input. When evaluating over time instead of number of function evaluations, we can approximately convert the time range T to a range of function evaluations F as shown in Eq. (3.10), where m_f is the mean time per function evaluation of Section 3.3.3.

$$F = \forall t \in T \left\lfloor \frac{t}{m_f} \right\rceil \tag{3.10}$$

An example of the visualization of an individual search space over time is shown in Fig. 3.5. Compared to Fig. 3.4, time is on the x-axis instead of function evaluations. The curves of the optimization algorithms look different over time, even though the general trajectories are the same.

So far, we have treated configurations as having a single unit of time. However, there are several costs in terms of time that can mask overhead (discussed in Section 3.2.1), which can lead to misrepresentation in results. We suggest providing additional analysis where necessary by breaking down the time spent on a single configuration into the following basic categories:

1. *Optimization algorithm time:* time taken by an optimization algorithm to select the next parameter configuration to try.
2. *Compilation time:* time taken to compile a single parameter configuration kernel.
3. *Run time:* The total time spent on kernel execution (i.e. all the runs of a single compiled parameter configuration, often executed multiple times to reduce noise).
4. *Result validation time:* Depending on the application and the tuning space, it might be necessary to validate the results of each kernel configuration, as some parameters might affect the program outcome validity. If that is the case, it is informative to report this time.
5. *Framework time:* All the other overhead costs involved in the tuning process (e.g. input/output operations, validation of configurations).

Note that this is not an exhaustive list. It may be relevant to break down the measurements further in analysis, for example, if communication dominates the *framework time*, the framework time can be further decomposed. While it is possible to apply the metrics of this section to these measurements individually, in practice, it is hard to analyze these completely separated in a concise, valid, and intuitive manner. Instead, we suggest selecting several relevant points in the time range to be analyzed. To mitigate cherry-picking, it is important to keep these points consistent, for example by selecting the 50th and 75th percentiles of the time range. This provides the necessary nuance to avoid masking overhead and facilitates further analysis into differences in scalability.

While accurate comparison of optimization algorithm performance on a single search space is helpful, the largest benefit is in comparing across search spaces over time, as this allows for conclusions regarding the overall performance of optimization algorithms. Note that it is not possible to aggregate across search spaces with function evaluations, as function evaluations are directly related to the characteristics of search spaces. Instead, time can be used to compare algorithm performance across search spaces. Therefore, the

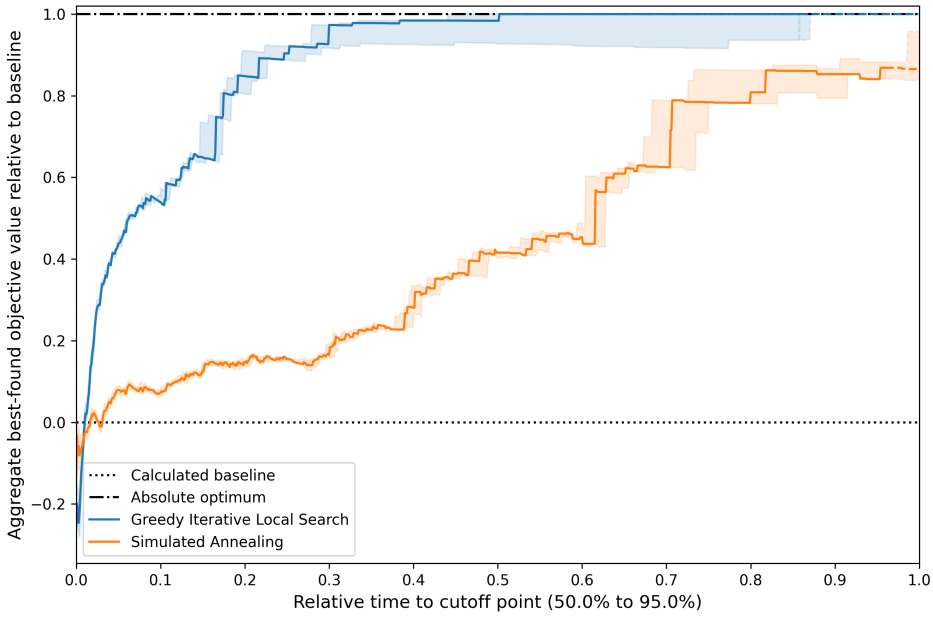
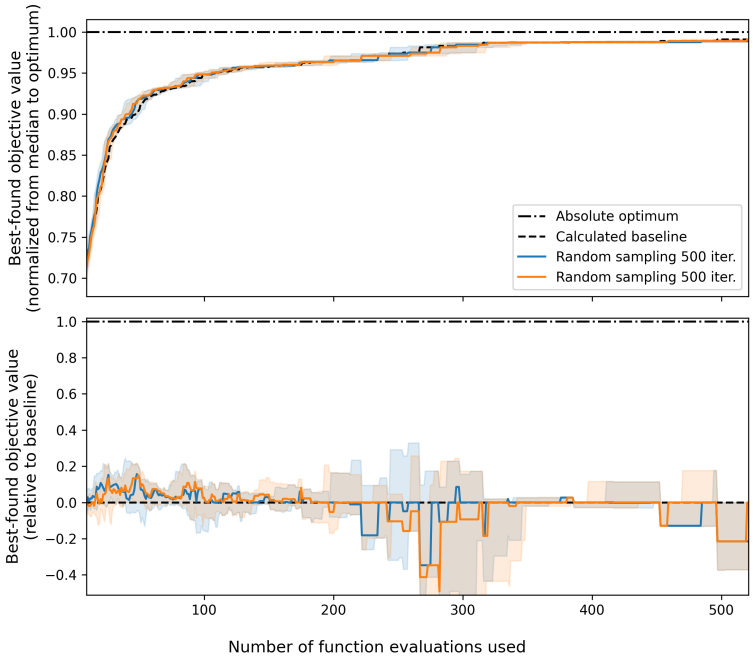


Figure 3.6: Example visualization of aggregate optimization algorithm performance across search spaces.

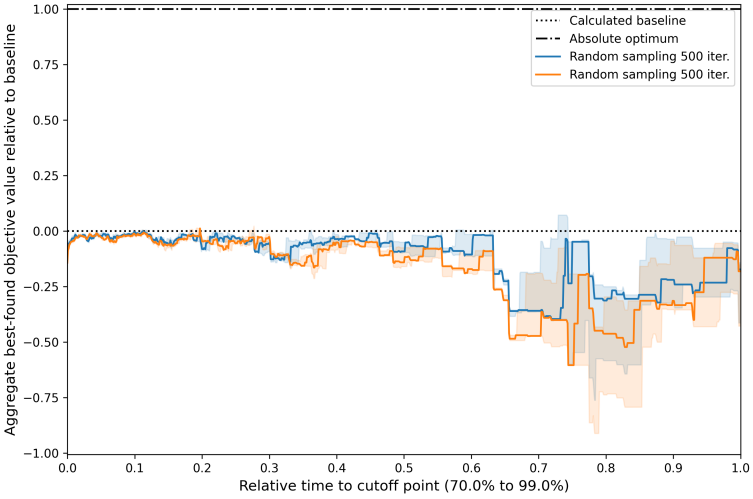
range of time must be the same for each search space. As the tuning budget of Section 3.3.3 is already relative to each search space, the time range can be made comparable across search spaces by normalizing from 0 at the start to 1 at the end of the tuning budget. If the start of this range is not of interest for the visualization, e.g. if all optimization algorithms take some time before finding better than random configurations, the tuning budget calculation of Section 3.3.3 can be used to calculate a starting cutoff point as well, for example to create a range from the time at $p = 0.5$ to the time at $p = 0.99$.

An example of the visualization of performance across search spaces is shown in Fig. 3.6. The most notable difference compared to the example of visualization for an individual search space (as in Fig. 3.5) is that instead of absolute time on the x-axis, this time is now relative to the cutoff point.

To validate the calculated random search baseline, Fig. 3.7 compares the expected approximate random search performance against Kernel Tuner's random search. The Kernel Tuner random search is shown with two trajectories of 500 random search runs each to show the sensitivity to randomness, even over such a large amount of iterations. A cutoff point of 99% is used, as even with random search very close to the global optimum we



(a) Performance over function evaluations for *convolution* on the RTX 3090.



(b) Aggregate performance for *convolution* and *GEMM* on the RTX 2080 Ti & RTX 3090.

Figure 3.7: Comparison of expected approximate random search performance against Kernel Tuner random search.

want to know if the calculated random search still matches the actual random search. It is important to note that, given the inherently random behavior of random search, the calculated random search is an approximation of this behavior.

In both Figs. 3.7a and 3.7b, the calculated random search baseline performs approximately similarly to both random search trajectories. However, there are some notable discrepancies. In both figures, the magnitude of the difference between the calculated baseline and runs increases as the number of function evaluations and time increases. This is because the distance between the global optimum and the baseline becomes increasingly smaller. This effect is an inherent limitation of this method and the reason the top plot is included, but can be mitigated by increasing search space size. Another discrepancy is the fact that in Fig. 3.7b, the executed random trajectories are always a bit worse than the calculated baseline. This is because the calculated baseline cannot take strategy time into account as this would make it implementation-dependent, while in practice even a simple random search will spend some time in this category.

So far, the assumption has been that the performance of optimization algorithms can be captured completely in a time range up to the tuning budget. However, some optimization algorithms may in practice stop before reaching the allotted tuning budget.

When this happens, the performance over time of an optimization algorithm should be split into two visually distinct parts, the real and the fictional part, which are split at the time more than 50% of the repeats have stopped searching. This is to avoid an early stopping algorithm where just a few runs have reached the maximum tuning budget to present this as if all have achieved this, drawing conclusions with too little data. For example, if an optimization algorithm gets stuck in a local optimum and stops searching before the budget is spent in most cases, yet in some cases it spends the entire budget and performs better, these few instances should not be presented as the performance that is to be expected. Instead, the real part shows the best-found average performance over the repeats, while the fictional part is simply set to the last best-found average performance of the real part.

This means that in the fictional performance part, the optimization algorithm will not be shown to find any improvements, and given enough time eventually be overtaken by all other optimization algorithms and random search, depending on the performance at the time of sampling. This is seen in among others Fig. 3.11, where after approximately 230 seconds the *genetic algorithm* stops and is subsequently surpassed by *dual annealing*. For aggregated performance, the mean fraction of the tuning budget at which an optimization

algorithm has stopped can be used to denote this distinction. This is visible in Fig. 3.6, where both optimization algorithms exhibit early-stopping behavior.

Finally, it is convenient to quantify these performance differences between optimization algorithms to be able to make global comparisons. The aggregate performance of an optimization algorithm offers various possibilities to do this. A simple yet powerful quantification is simply taking the mean over time of the aggregate performance score. A complication could be early-stopping algorithms that perform very well before stopping. Taking only the real part into account would then provide unrealistically positive expectations of the performance of such an algorithm. As such, the fictional part should be included in this mean as well. This mean over time of the aggregate performance score allows, for example, a quick comparison between optimization algorithms in different papers, for which the raw data to compare the aggregate performance is not directly available to the reader.

3.4 Application & Evaluation

In this section, it is first described how the methodology can be easily applied using the software package (Section 3.4.1), and evaluated in an application case study intended to serve as a demonstration (Sections 3.4.2 to 3.4.5). We will attempt to answer the showcase research question *"Which of various optimization algorithms in Kernel Tuner and Kernel Tuning Toolkit (KTT) is best suited to auto-tuning GPU kernels?"*. This case study is executed in line with the steps of the methodology as described in Section 3.3. For the sake of brevity, it is not possible to be as detailed as the authors would and should be in an independent publication; this is intended to be a demonstration for evaluating the methodology, not a comparison of the optimization algorithms involved. Finally, it is discussed how the potential discrepancies of Section 3.2.1 and the methodological limitations observed in practice in Section 3.2.2 have been addressed or mitigated using this methodology in Section 3.4.6.

3.4.1 Application using the software package

Section 3.3 described the methodology in detail. However, this is an extensive description that would be infeasible to request each author to manually implement for each publication. As such, a software package is published alongside this chapter that provides the implementation of this methodology.

The software is implemented in Python and can be easily configured for multiple experiments via JSON configuration files. The experiment part can run separately from the visualization part, making it possible to execute an experiment on one system (e.g. a cluster) and visualize it on another (e.g. a local machine). With an advanced caching system built-in, it is easy to reproduce experiments at a later moment. It also supports an interactive mode for use in notebooks.

This software has also been used to produce the plots seen in this chapter, for which the documentation[2] and repository[3] are publicy available. The package can be installed with `pip install autotuning_methodology`.

3.4.2 Step 1: Experimental Setup

As mentioned in the research question, we will focus on various Kernel Tuner and KTT algorithms. The latest versions at the time of writing were used, respectively 1.0 and 2.1. As GPUs, we will use the NVIDIA RTX 2080 Ti and RTX 3090. The RTX 2080 Ti uses CUDA 12.1, driver 530.30.02, on Ubuntu 18.04, paired with an Intel i7-8700 and 32GB of DDR4 RAM. The RTX 3090 uses CUDA 12.2, driver 535.171.04, on Ubuntu 20.04, paired with an AMD EPYC 7402 and 64GB of DDR4 RAM.

The previously mentioned BAT benchmark by Tørring, van Werkhoven, Petrovč, Willemsen, Filipovič, and Elster [159] will be used, as this benchmark has taken balanced search spaces of appropriate size, parameter influence, and appropriate values into account. As this is a simple demonstration, we only use the *convolution* and point-in-polygon (*pnpoly*) applications from this benchmark, whereas normally the entire benchmark suite would be used. For the *convolution* kernel, an image width and height of 4096 is used with a filter width and height of 15. For the *pnpoly* kernel, 2e7 points are used.

3.4.3 Step 2: Optimization Algorithms

To demonstrate the capability of the methodology in comparing optimization algorithm performance across kernels and GPUs, we have selected a diverse set of state-of-the-art optimization algorithms [134, 158, 48] across the auto-tuning frameworks Kernel Tuner and KTT that generate and traverse search spaces in a variety of ways. From Kernel Tuner, the following optimization algorithms are used:

- Greedy Iterative Local Search, a relatively simple local search algorithm that is iteratively initiated to optimize globally. It has four hyperparameters: the neighbor method (default Hamming distance), whether to restart from a position as soon as

[2] www.autotuningassociation.github.io/autotuning_methodology
[3] www.github.com/AutoTuningAssociation/autotuning_methodology

an improvement is found (default true), the number of evaluations without improvement before restarting (default 50), and a random factor to restart from a position as soon as an improvement is found (default 0.3).

- Genetic Algorithm, an evolutionary algorithm that emulates natural selection and mutation. It has four hyperparameters: the population size (default 20), the maximum number of generations (default 100), the crossover method (default uniform), and the mutation rate (default 10).

- Dual Annealing, a hybrid of simulated annealing and differential evolution to explore globally and exploit locally. The Kernel Tuner implementation has a single hyperparameter, the local optimization method (default Powell), and uses SciPy 1.10.1.

From KTT, the profile-based searcher [48] is used, which collects performance counters during the kernel execution and feeds these to a model trained on previously obtained data to guide the search. To mimic a scenario where developers build the model on their hardware and users use the model to do auto-tuning on other hardware, the model is trained using data from a different GPU architecture. The profile-based searcher has four main hyperparameters: the batch size (default 5), the number of configurations neighboring the previously profiled configuration to evaluate (default 100), the number of non-neighboring configurations to evaluate (default 10), and the maximum number of dimensions allowed to vary between neighbors (default 2). In this work, the model used on the RTX 2080 Ti was trained on the RTX 3090 (Ampere), and the model used on the RTX 3090 was trained on the RTX 2080 Ti (Turing). The time taken to obtain the data and train the model is not included in the times of the methodology. However, other searchers allow broader usage as they work out of the box without the need for any experiments prior to autotuning.

As this is simply a showcase of the methodology we will not apply hyperparameter tuning, instead using the defaults of each optimization algorithm.

3.4.4 Step 3: Tuning Budget and Noise

The tuning budget for these experiments is set to 96%, meaning that the distance between the objective value of the median configuration and the global optimum is covered up to 96%. The starting cutoff point (as described in Section 3.3.4) is set to 50%. The resolution of the ranges (as required for visualization over time) is set to 1000. The individual configurations are all sampled 32 times each, while the optimization algorithms are run 100 times each.

3.4.5 Step 4: Quantifying Performance

Given the setup of this simple showcase experiment, the metrics can now be applied to find out which optimization algorithm performs best in these circumstances.

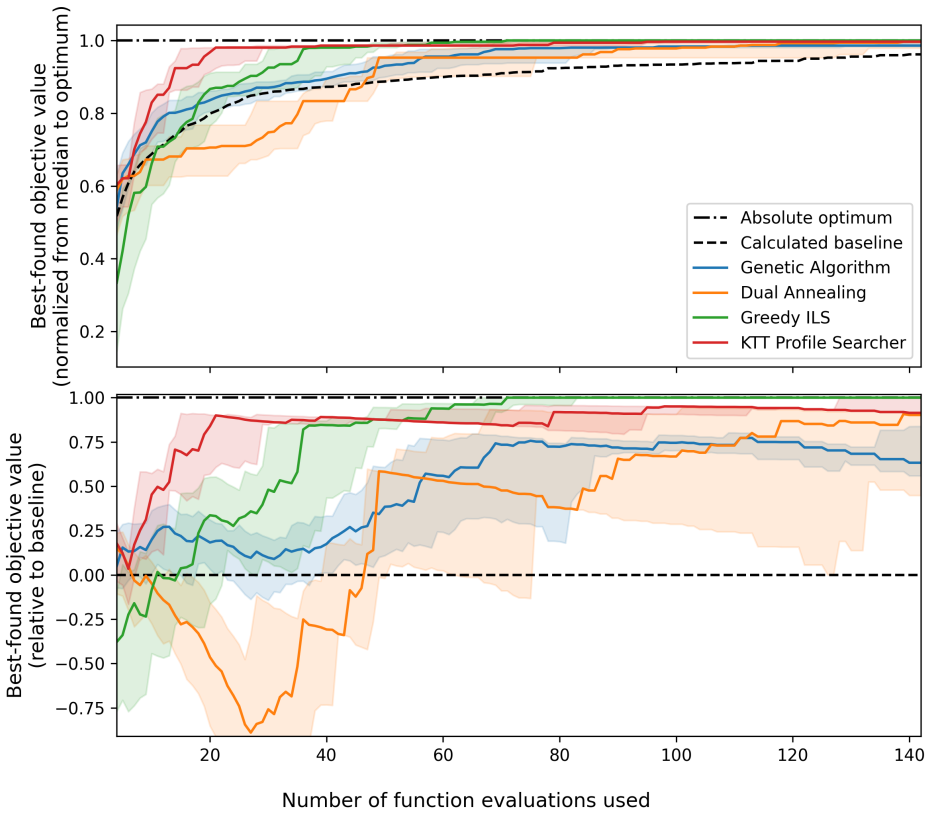


Figure 3.8: Performance over function evaluations for the *pnpoly* kernel on the RTX 2080 Ti.

Algorithm	Compilation	Benchmark	Framework	Searcher
Genetic Algorithm	0.115	0.000755	2.89e-05	2.92e-06
Dual Annealing	0.118	0.000829	0.000425	8.98e-05
Greedy ILS	0.125	0.000804	2.75e-05	4.04e-06
KTT Profile Searcher	1.24	0.00867	0.0322	0.00769

Table 3.2: Median split times in seconds per configuration.

We start with the application on the individual search spaces, as shown in Figs. 3.8 and 3.9. The *pnpoly* application is shown with both number of function evaluations (Fig. 3.8) and time (Fig. 3.9). This demonstrates the difference in reported performance between two

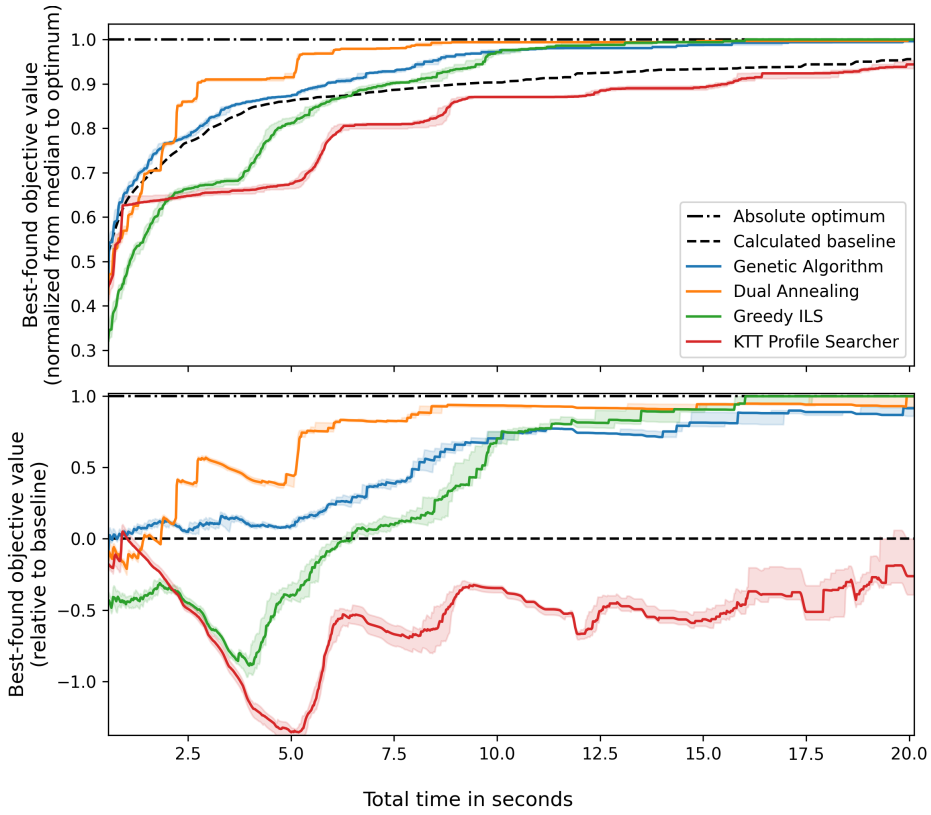


Figure 3.9: Performance over time for the *pnpoly* kernel on the RTX 2080 Ti.

measures of time: with number of function evaluations in Fig. 3.8, the *KTT profile searcher* and *Greedy ILS* appear to perform very well, but with time taken into account in Fig. 3.9, it is actually *dual annealing* that performs best overall.

Looking at Table 3.2, the compilation time dominates. As the *KTT profile searcher* profiling time is counted towards the compilation category, the large difference in performance between Fig. 3.8 and Fig. 3.9 for this algorithms is explained. As explained in Section 3.2.1, the specifics of a kernel and searchspace can mask or emphasize certain aspects of an algorithm that are not necessarily true in general.

Looking at the performance of the optimization algorithms on the other search spaces in Figs. 3.10 to 3.12 demonstrates another advantage of this methodology: while literature often shows plots similar to the normalized subplot shown at the top of for example Fig. 3.11, it is considerably easier to gain insight in the performance of optimization algorithms in the subplot below with random as a baseline. In addition, Fig. 3.11 also shows that *genetic algorithm* does not utilize the full tuning budget on this searchspace, resulting

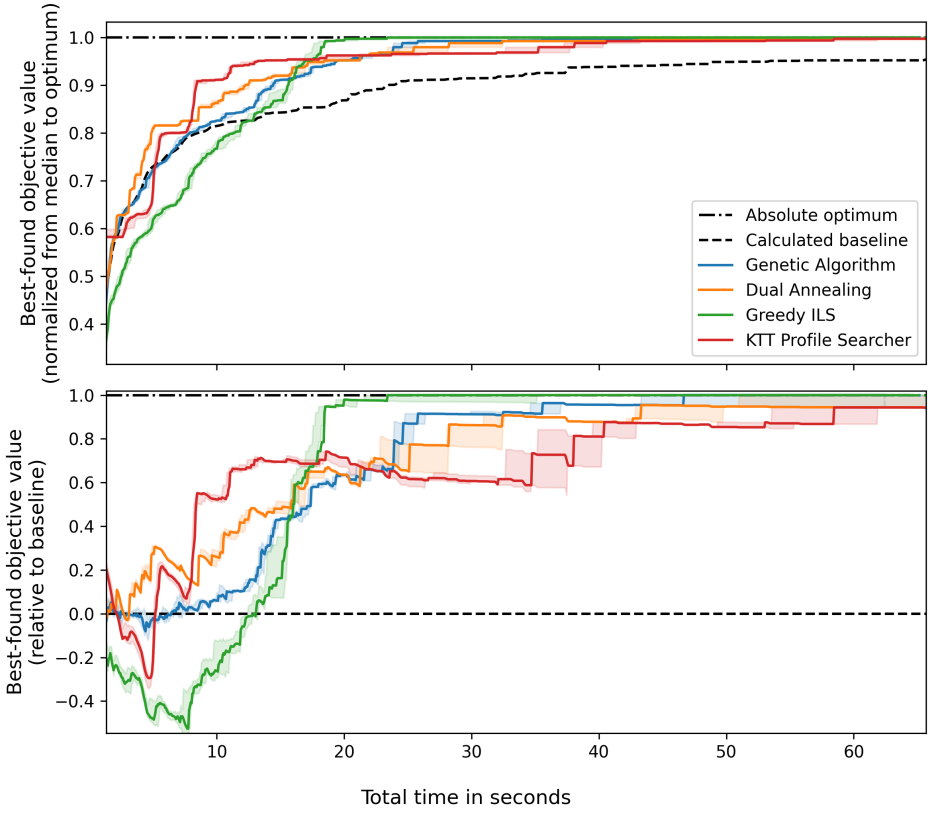


Figure 3.10: Performance on *pnpoly* on the RTX 3090.

in the dashed fictional parts for that algorithm at approximately 260 seconds. The advantage of showing both the normalized and relative to baseline trajectories is also demonstrated. For example, in Fig. 3.11, *Greedy ILS* barely finds any configurations better after approximately 190 seconds than what it has found before that time, while the baseline continuously finds better configurations. Hence, the relative performance decreases.

Algorithm	Performance score	Standard deviation
Genetic Algorithm	0.537	0.308
Dual Annealing	0.599	0.267
Greedy ILS	0.352	0.403
KTT Profile Searcher	0.49	0.146

Table 3.3: Aggregate performance scores per optimization algorithm.

While from Figs. 3.9 to 3.12 it can be said that *Greedy ILS* appears to be worst overall, these plots demonstrate that even with the same time unit, it is hard to come to conclusions,

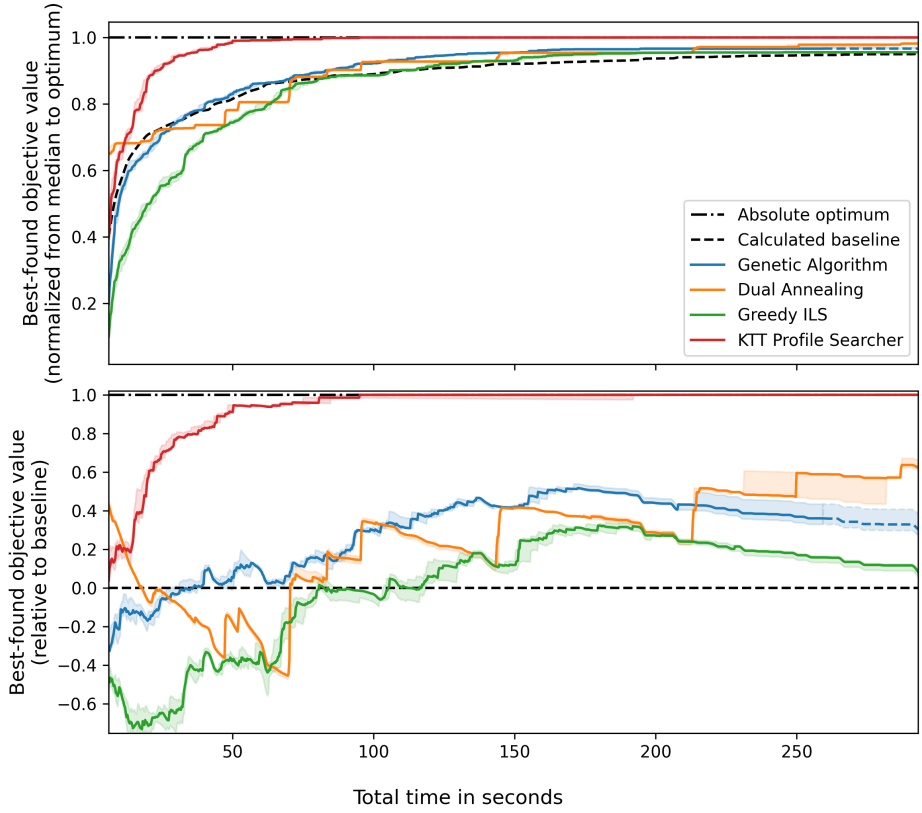


Figure 3.11: Performance on *convolution* on the RTX 3090.

and the difference in time scale between the figures adds to the difficulty. For example, there are counterpoints to the statement that *Greedy ILS* is the worst: in Fig. 3.10 it is the first to find the optimum. In contrast, the aggregate performance across the *pnpoly* and *convolution* kernels and the RTX 2080 Ti and RTX 3090 displayed in Fig. 3.13 gives a clear picture of this: in general, *Dual Annealing* ranks best towards the end, while the *KTT profile searcher* performs best at the start, *genetic algorithms* is stable middle ground, and *Greedy ILS* performs worst. This is confirmed by the aggregate performance scores seen in Table 3.3.

3.4.6 Discussion: ”Six ways to fool the masses” and Limitations

We recall the six ways performance results of optimization algorithms in auto-tuning can be influenced as described in Section 3.2.1, as well as the methodological limitations observed in Section 3.2.2, and briefly describe how the methodology has addressed or mitigated those concerns.

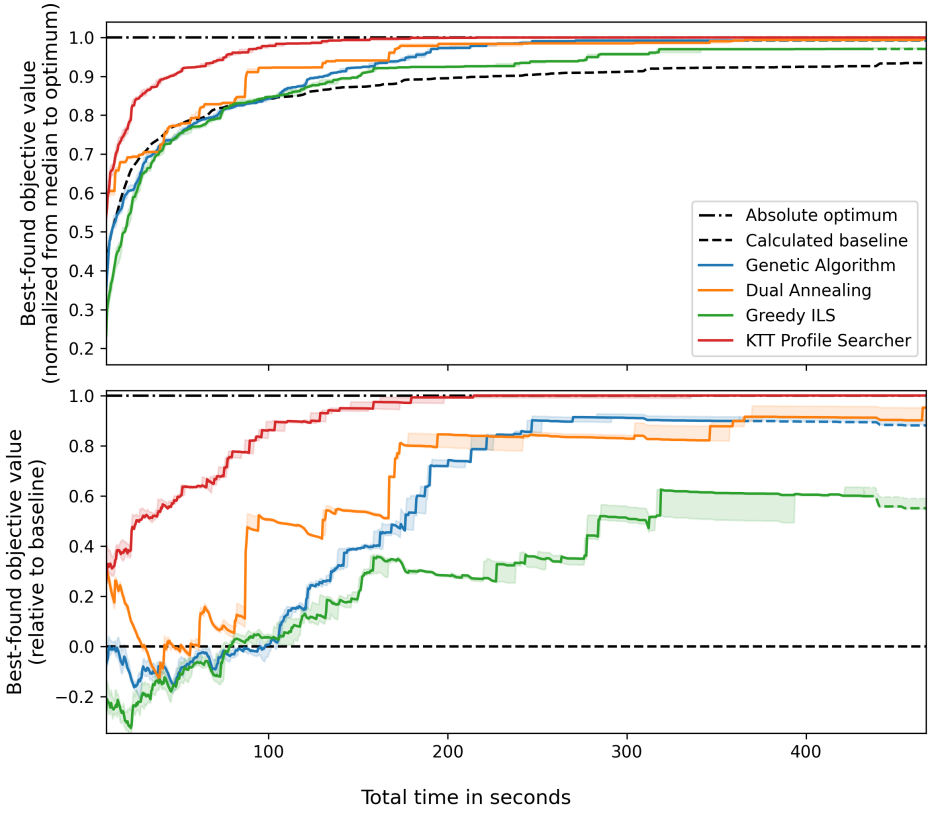


Figure 3.12: Performance on *convolution* on the RTX 2080 Ti.

The use of a fixed set of kernels in a well-designed benchmark suite addresses the issues raised regarding changing kernel run-time (Section 3.2.1), emphasizing or masking overhead (Section 3.2.1) and degenerating the search space (Section 3.2.1). This also prevents the use of too few or even single kernels and small search spaces observed in practice in Section 3.2.2. Metrics such as Permutation Feature Importance and Proportion of Centrality have been successfully applied to ensure tunable parameters significantly impact performance and search space difficulty can be quantified, leading to well-formed configurations in the search spaces.

Section 3.2.1 described how results can be distorted by selecting or defining favorable metrics. In addition, As the methodology states which metrics are to be used and provides a strict definition of these metrics, this is avoided, which also addresses the ”reporting” category of Table 3.1. The value of this can be seen in Section 3.4.5, where a large difference between performance over function evaluations and time is observed.

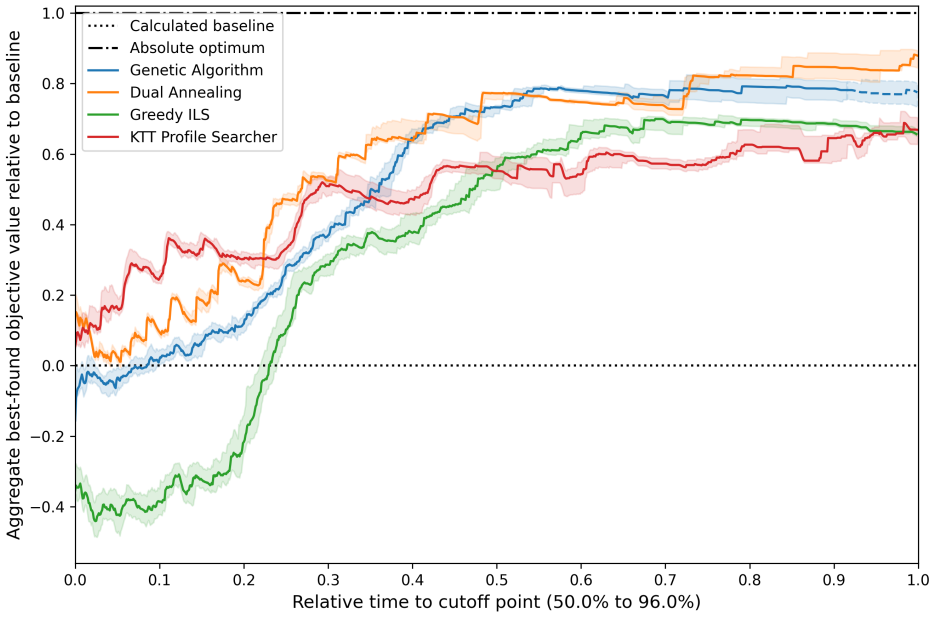


Figure 3.13: Visualization of optimization algorithm performance aggregated across both kernels and both GPUs.

While the fact that results can be influenced by adjusting the environment as described in Section 3.2.1 is particularly hard to detect or mitigate, requiring the use of multiple, diverse search spaces makes it hard to actively manipulate results. In addition, this mitigates the pitfall of reporting results on too few or even a single device and the underspecification of environments commonly observed in Section 3.2.2.

Finally, Section 3.2.1 described how ambiguity regarding the implementation used can be of influence, which is also observed in practice in the ”reproducibility” category of Table 3.1. An important mitigating factor provided by the methodology is that the baseline is implementation-independent, meaning that if a less efficient implementation of an optimization algorithm is used, this can be easily detected by comparing it to another implementation using the baseline.

3.5 Related Work

This related work section focuses on how other fields have implemented methodologies for the comparison of optimization algorithm performance.

In 1997, Billups, Dirkse, and Ferris [21] proposed to use the ratio of an optimization algorithm against the best-performing optimization algorithm, and group into arbitrary

categories based on performance relative to the best-performing optimization algorithm. This enables moving towards a large, representative benchmark where the aggregates are more important than the exact individual scores. Dolan and Moré [43] overviewed common metrics, highlighting each method's strengths and weaknesses. Extending Billups, Dirkse, and Ferris [21], they introduced performance profiles as cumulative distribution functions for optimization algorithms based on a performance metric. The performance metric, based on the ratio of problems solved within a factor of the best optimization algorithm time, ensures stable scoring even with a large test set. This stability reduces the impact of outliers or noise, accurately representing expected performance across problems. Later on, data profiles were introduced by Moré and Wild [109] to capture short-term optimization algorithm behavior, allowing for analysis of optimization algorithm performance in cases of constraint computational budget.

Despite becoming *"a gold standard in benchmarking of optimization algorithms"* [17], performance profiles have limitations. Key issues include reliance on arbitrary convergence test definitions and the absence of a total ordering among all algorithms [59]. Another limitation of performance profiles is their dependency on a 'best' optimization algorithm for aggregating performance data.

Pelikan [119] conducted pairwise comparisons of several genetic and evolutionary algorithms on randomly fitness landscapes. Goldman and Punch [57] instead compared optimization algorithms using the median number of function evaluations to reach the optimum. Mersmann, Preuss, and Trautmann [103] propose a consensus-based metric for ranking algorithms on noiseless functions, stating that a consensus-based metric is needed as not all desirable criteria may be satisfied all the time.

Brownlee [25] highlights the absence of a consistent, meaningful benchmarking approach in optimization algorithms, concluding that no universal method exists for empirical evaluation and comparison, akin to the 'no free lunch' theorem. Hooker [73] argues against competitive testing of heuristic algorithms, advocating for the investigation of their behavior instead.

The current state of methodology and metrics for optimization algorithms are perhaps best captured by Beiranvand, Hare, and Lucet [17], who provide best practices based on Dolan and Moré [43], while taking into account the known limitations. They argue that the reason for benchmarking and what aspect of the algorithm is most important are often neglected, while these factors are essential in determining the value of the results. They further discuss the choices and challenges in benchmarking, offering a table with several performance measures based on performance categories. Interestingly, they argue all algorithms should use the same starting points. This is, however, not realistic for auto-tuning,

where the initial sampling affects optimization algorithm performance. On the issue of hyperparameter tuning and fair comparison of algorithms, it is stated that either none or all solvers should have their hyperparameters tuned. They argue that while there are several techniques for visualizations of individual problems, this does not work for aggregation across problems. Instead, the performance profiles seen in [43] are used to graphically indicate optimization algorithm performance across test problems.

Bartz-Beielstein, Doerr, Berg, Bossek, Chandrasekaran, Eftimov, et al. [15] collects best practices in optimization algorithm benchmarking using a survey in a periodically updated manuscript, first published in 2020. They consider eight topics essential to any benchmark study: "*goals, problems, algorithms, performance, analysis, design, presentation, and reproducibility*", and propose a rudimentary checklist based on these topics, to be expanded in the future.

There are also several frameworks and tools to automate optimization algorithm evaluation. For example, Hansen, Auger, Ros, Mersmann, Tušar, and Brockhoff [61] presents COCO, an open-source platform for comparing continuous optimization algorithms in a black-box setting. Doerr, Wang, Ye, van Rijn, and Bäck [42] introduce IOHprofiler, a benchmarking and profiling tool for iterative optimization heuristics building on COCO. Such ready-to-use tools implementing a methodology can ease the adoption within a community.

Finally, as perhaps the most closely related work, Tørring and Elster [158] demonstrates how the impact of tuning budget choices in comparing auto-tuning optimization algorithms makes accurate comparisons difficult. The performance of several optimization algorithms is shown with various tuning budgets, demonstrating that there is no algorithm that performs optimally across all budgets. Nevertheless, the sensitivity of the results to the characteristics of evaluations demonstrates the need for statistically sound evaluation methods.

3.6 Conclusions

In this chapter, we propose a methodology for comparing optimization algorithms in auto-tuning. Methodological limitations in auto-tuning were discussed by summarizing the current state of the practice and discussing scenarios in which performance results can be distorted. Based on this and similar work from related fields, we proposed a methodology that addresses these concerns. The capabilities of this methodology were evaluated on a test case of four search spaces, which demonstrated the benefits. Nevertheless, there are several remarks regarding the limitations of this work.

We have focused mainly on GPU auto-tuning, even though the auto-tuning field is broader than GPUs. Nevertheless, the applicability of this methodology is not limited to GPUs. It is important to note that the methodology can also be applied if the optimization objective is not performance.

As the methodology uses random search as a baseline, the search spaces must be well-formed. It is thus important to apply the methodology using a benchmark that adheres to the discussed standards and characteristics. It can be necessary to re-evaluate this when the benchmark is applied in new conditions, such as a new architecture. Search spaces need to be fully explored (brute-forced) to have the statistical underpinnings required. This also has upsides; once a search space has been fully explored, given that a simulation mechanism is in place, flexibility is increased and it is more efficient to compare optimization algorithms. Complications and opportunities in applying this methodology to search spaces too large to brute-force are future work.

This research makes several contributions: it provides an overview of the state of the auto-tuning practice, establishes best practices regarding the setup, execution, and presentation of auto-tuning research, and enables fair comparisons over the true time spent. In addition, our methodology can easily be applied by others via the supplied open-source software package[4]. These contributions together allow researchers in auto-tuning to apply this methodology to improve the field in general.

[4] www.github.com/AutoTuningAssociation/autotuning_methodology

4 Efficient Construction of Large Search Spaces

> *Perfection is achieved, not when there is nothing more to add, but when there is nothing left to take away.*
>
> Antoine de Saint-Exupéry, *Airman's Odyssey*, 1942

For many auto-tuning tasks, constructing the search space itself can be a major bottleneck, taking minutes to days due to combinatorial explosion and complex constraints. We reformulate search space construction as a Constraint Satisfaction Problem (CSP) and develop a runtime parser to translate user constraints into solver-optimized forms. By exploiting common constraint structures and integrating targeted solver optimizations, we achieve large performance gains while preserving flexibility. Benchmarks show speedups of up to four orders of magnitude over brute force, and one to two orders over chain-of-trees–based frameworks. Our open-source solution removes a key scalability barrier for auto-tuning and related domains.

This chapter is based on "Efficient construction of large search spaces for auto-tuning" by Willemsen, van Nieuwpoort, and van Werkhoven [171].

4.1 Introduction

At the heart of the auto-tuning method is a search space of functionally-equivalent *code variants* that is explored by an optimization algorithm. These code variants can be generated by a compiler or using metaprogramming techniques, such as application-specific code generators or function templates. Together, these code variants constitute vast search spaces that are infeasible to search by hand [133, 136, 114] and would have to be searched over and over again as the application is executed on different hardware or different input data sets and sizes [115, 166, 84, 121]. Construction of the auto-tuning search space, used for enumerating and sampling different code variants, is a crucial factor in the performance of auto-tuning as search space sizes increase, and has therefore received a lot of attention recently [125, 126, 68, 135].

The difficulty in constructing search spaces for auto-tuning arises from the fact that not all code variants constitute valid implementations. In fact, many code variants that could potentially be generated violate so-called *constraints*. The constraints formulate dependencies between different tunable parameters in the code and often depend on limitations in both the program and the hardware. For example, when applying loop blocking, the tile size of the outer loop has to be a multiple of the tile size used in the inner loop, or a padding scheme in shared memory that only applies when shared memory is used and only for certain thread block dimensions.

In modern applications, where search spaces may contain millions or even billions of configurations, constructing the search space can take minutes to days, making search space construction a major bottleneck [159, 67, 68]. This is problematic when auto-tuning for a single target, but even more prohibitive on a diverse set of target input data and hardware, such as a BLAS library.

To this end, Rasch et al. [125] have introduced a method, referred to as *chain-of-trees*, specifically designed for the purpose of efficiently constructing search spaces for constraint-based auto-tuning. The chain-of-trees approach starts with identifying groups of interdependent parameters. Two parameters are interdependent if they both occur in the syntax tree of the same constraint descriptor. For each parameter group, a tree is constructed that encodes all possible combinations of interdependent parameter values. Finally, the trees are linked together to form a chain-of-trees. The chain-of-trees method is widely considered to be state-of-the-art and has been integrated into several auto-tuning frameworks, including ATF [126], BaCO [68], PyATF [135], and KTT [120].

In this chapter, instead of adopting a customized solution for constraint-based auto-tuning, such as chain-of-trees, we investigate the use of methods with a more robust mathematical foundation. In particular, we show that the problem of search space construction

in constraint-based auto-tuning can be automatically reduced to a Constraint Satisfaction Problem (CSP). To enable the use of CSP in a state-of-the-art auto-tuner, we employ various techniques, including run-time compilation to translate user-defined constraint functions to representations that can be used directly in existing CSP solvers, as well as major improvements to the performance of such solvers. We evaluate the CSP and chain-of-trees methods on a wide variety of search spaces, including the search spaces used by Rasch et al. [126]. Evaluation shows that our optimized CSP-based search space construction method is four orders of magnitude faster than brute force construction, three orders of magnitude faster than an unoptimized CSP solver, and one to two orders of magnitude faster than other state-of-the-art auto-tuning frameworks.

Our work has been integrated into the auto-tuning framework Kernel Tuner [165] and a Python-based CSP solver named python-constraint, both existing open-source projects with a substantial number of users, to benefit both communities.

The rest of this chapter is structured as follows. Section 4.2 provides a high-level introduction to the role of constraints in auto-tuning. Section 4.3 discusses related work. Section 4.4 describes the design and implementation of our method. In Section 4.5, we evaluate the efficiency and scalability of our optimized CSP-based approach against various state-of-the-art solutions, and Section 4.6 concludes.

4.2 Constraint-based Auto-tuning

This section provides a general introduction to auto-tuning using an example kernel to illustrate how constraints arise when creating tunable applications for modern highly parallel architectures.

We will use the Hotspot kernel from the BAT [159] benchmark suite of tunable kernels as an example. This Hotspot kernel, adapted from the Rodinia Benchmark suite [27], simulates heat dissipation in a microprocessor based on the processor's architectural floor plan, thermal resistance, ambient temperature, and simulated power currents. Listing 1 shows the Hotspot kernel code in HIP/CUDA, simplified for readability by removing bound checks.

The kernel uses a two-dimensional thread block to calculate the temperature of the chip at the new time step, where each thread computes one value in the output matrix `Tout`. In this kernel, we can change the thread block x and y dimensions without affecting the output, as long as we create enough thread blocks to cover the entire problem domain. In other words, the thread block dimensions in x and y are *tunable parameters* that affect the performance but not the outcome. The optimal values for these parameters are highly specific to the kernel, the target hardware platform, and the input problem. We thus select a wide range of values for both parameters.

```
__global__ void calculate_temp(float **Tout, float **Tin, float
    Tambient, float **Power, float3 R, float dt) {
  int x = blockIdx.x * blockDim.x + threadIdx.x;
  int y = blockIdx.y * blockDim.y + threadIdx.y;

  Tout[y][x] = Tin[y][x] + dt * ( power[y][x] +
        (Tin[y+1][x  ] + Tin[y-1][x  ] - 2.0*Tin[y][x]) * R.y +
        (Tin[y  ][x+1] + Tin[y  ][x+1] - 2.0*Tin[y][x]) * R.x +
        (Tambient - Tin[y][x]) * R.z);
}
```

Listing 1: Example Hotspot kernel in HIP/CUDA.

However, to ensure sufficient parallelism, we need to make sure that the thread block contains at least 32 threads. In addition, most parallel architectures pose an upper bound on the number of threads per block, which can be queried before tuning. For simplicity, we here assume that this limit is 1024 threads. These restrictions on the two parameters together form a constraint, namely

`32 <= thread_block_x * thread_block_y <= 1024`.

Different autotuners support different formats for specifying constraints, as illustrated in Listing 2. ATF and PyATF both define constraints directly on the last parameter of a group of interdependent parameters, whereas KTT and Kernel Tuner define constraints separately from the tunable parameters. Kernel Tuner allows constraints to be defined as lambda functions or using string expressions. The model that all tuners follow is that the lambda functions, or string expressions, should evaluate to `True` for any specific combination of tunable parameter values to be considered a *valid* candidate solution, or code variant, in the search space.

The constraint defined in Listing 2 only involves the thread block dimensions. However, the fully optimized version of the Hotspot kernel contains many more tunable parameters and constraints. For example, the number of output elements computed by each thread can be varied in both the x- and y-dimensions, introducing two more tunable parameters. Another tunable parameter (`sh_power`) controls whether or not to cache the `Power` values in *shared memory*. Together, these parameters form another constraint: `(thread_block_x * work_per_thread_x) * (thread_block_y * work_per_thread_y) * sh_power * 4 <= max_shared_memory_per_block` to avoid exceeding the maximum amount of shared memory allowed per block in bytes. The full Hotspot kernel is even more complex and also implements temporal tiling, partial loop unrolling, and double buffering of the temperature field in shared memory, resulting in several more complex constraints. For further specification of the kernel see [159].

```
// KTT
auto minWGConstraint = [](const std::vector<uint64_t>& v) {return v[0] * v[1] >= 32;};
tuner.AddConstraint(kernel, {"thread_block_x", "thread_block_y"}, minWGConstraint);
auto maxWGConstraint = [](const std::vector<uint64_t>& v) {return v[0] * v[1] <= 1024;};
tuner.AddConstraint(kernel, {"thread_block_x", "thread_block_y"}, maxWGConstraint);

// ATF combines tunable parameter and constraint declaration
auto thread_block_y = atf::tuning_parameter("thread_block_y", atf::interval<size_t>(1, N), [&](size_t thread_block_x){
    return (thread_block_x*thread_block_y >= 32 &&
    thread_block_x*thread_block_y <= 1024) });

// PyATF uses an interface similar to ATF, but in Python
thread_block_y = TP('thread_block_y', Interval(1, M), lambda thread_block_y, thread_block_x: thread_block_x*thread_block_y >= 32 and thread_block_x*thread_block_y <= 1024)

// Kernel Tuner (lambda-based constraint API)
constraint = lambda p: 32 <= p["thread_block_x"] * p["thread_block_y"] <= 1024

// Kernel Tuner (string-based constraint API)
constraint = "32 <= block_size_x*block_size_y <= 1024"
```

Listing 2: Example of constraints specification in different tuners.

4.3 Related Work

Table 4.1 shows an overview of support for specifying constraints, as well as the method used for search space construction in related auto-tuning frameworks. As we can see, some frameworks rely on brute force search space construction, *ytopt* and *GPTune* use `ConfigSpace` and `scikit-optimize.space` respectively, while the chain-of-trees method is most commonly used. We will briefly discuss the pros and cons of these different approaches.

In the absence of constraints, the search space is defined as the Cartesian product of all possible combinations of all tunable parameter values. In brute-force search space construction, the approach is to simply iterate through all possible combinations and filter out combinations that violate the constraints. This is reasonable for small search spaces, but becomes increasingly time-consuming as the search space size and number of constraints increase. For various auto-tuning applications, the number of valid configurations in the

Table 4.1: Overview of constraint support and search space construction methods in related work and this work (Kernel Tuner). ⋆ While ytopt and GPTune are actively maintained, dependencies *ConfigSpace* and *scikit-optimize* are not.

Tuner	Open Source	Actively developed	Constraints API	Search Space Construction
AUMA [47]	✓	✗	n/a	external
CLTune [114]	✓	✗	C++	brute-force
OpenTuner [3]	✓	✗	n/a	brute-force
ytopt [175]	✓	✓⋆	Python	ConfigSpace
GPTune [95]	✓	✓⋆	Python	scikit-optimize.space
KTT [120]	✓	✓	C++	chain-of-trees
ATF [125]	✓	✓	C++	chain-of-trees
BaCO [68]	✓	✗	JSON	chain-of-trees
PyATF [135]	✓	✓	Python	chain-of-trees
Kernel Tuner	✓	✓	Python	CSP solver

search space is orders of magnitude smaller than the Cartesian product size, causing the vast majority of generated and evaluated combinations to be discarded, wasting time and resources.

`ConfigSpace` and `scikit-optimize.space` are both Python packages that implement functionality to represent multidimensional configuration spaces. Both approaches do not enumerate or store individual configurations, but instead provide an interface to generate random samples from the search space. As constraint resolution is not supported by `scikit-optimize.space`, GPTune relies on an additional internal check on sampled points. While `ConfigSpace` does allow users to specify constraints, called *forbidden clauses* in `ConfigSpace`, these constraints are again only checked after generating the sample point. The advantage of this approach is its simplicity and ability for uniform sampling over all points in the unconstrained Cartesian space. However, the disadvantage of this approach is that constraints are not taken into account when generating samples, which has the same efficiency downsides as brute force search space construction.

Rasch et al. [125] introduced the chain-of-trees structure to represent constrained search spaces in constraint-based auto-tuning. Parameters are grouped based on interdependencies in the constraint functions, and each group is represented by a tree that encodes valid parameter combinations. These trees are then linked sequentially, forming a chain. This structure can reduce redundancy by reusing shared subtrees, which may lead to lower memory usage compared to flat representations. Independent parameters are handled as single-parameter trees.

In Section 4.4, instead of adopting a customized solution for constraint-based auto-tuning such as chain-of-trees, we investigate the feasibility of using established methods with a robust mathematical foundation for efficient search space construction.

4.4 Application and Optimization of CSP Solvers

In this section, we discuss the design and implementation details of our novel method for efficiently constructing large search spaces for constraint-based auto-tuning. Section 4.4.1 examines various constraint-solving techniques to formalize the relation with auto-tuning and find a robust solver best suited to the problem context. Section 4.4.2 describes the automatic optimization of user-defined constraints, followed by solver optimizations in Section 4.4.3 to improve efficiency. Finally, Section 4.4.4 details how the resulting search space is represented and applied in auto-tuning frameworks.

4.4.1 Using Constraint Solvers in Auto-tuning

The search space construction problem, where parameter values and constraints must be resolved to all valid combinations, can generally be encoded as a Boolean Satisfiability Problem (SAT) [20], Satisfiability Modulo Theories (SMT) [14], or Constraint-Satisfaction Problem (CSP) [23]. Among SAT, SMT, and CSP, the technique closest to auto-tuning search space construction is CSP. While SAT deals with Boolean variables and SMT extends SAT with theories like arithmetic or arrays, CSP solvers offer high-level abstractions suitable for modeling complex constraints that are otherwise difficult to efficiently express, making them better suited for modeling the complex relationships found in auto-tuning problems.

We can formalize the auto-tuning search space construction problem as a CSP defined by $\mathscr{P} = (X, D, C)$, where:

- $X = \{x_1, x_2, \ldots, x_n\}$ is a finite set of variables, each corresponding to a tuning parameter (e.g., block size, tile width).
- $D = \{D_1, D_2, \ldots, D_n\}$ is a set of finite domains, where D_i is the set of legal values for variable x_i.
- $C = \{c_1, c_2, \ldots, c_m\}$ is a finite set of constraints, where each c_j is a predicate over a subset of variables $\text{scope}(c_j) \subseteq X$.

A solution to the auto-tuning search space is then a total assignment $\mathscr{V} : X \to \bigcup D_i$ such that $\mathscr{V}(x_i) \in D_i$ for all i, and all constraints $c_j \in C$ are satisfied under $\mathscr{V}$. The overall auto-tuning objective is to determine the optimal configuration as

$v^\star = \arg\max_{v \in \mathcal{V}} f_{H_j,I_k}(A_{i,v})$, where we have an application A_i on a hardware platform H_j for an input dataset I_k to maximize performance $f_{H_j,I_k}(A_i)$ over the code variants in $\mathcal{V}$.

In addition, there are practical considerations when it comes to choosing a solver to build on. As we will implement our solver in the Python-based Kernel Tuner, the solver should be deployable in a Python environment. Notable options include *OR-Tools* [97] and the Z3 solver in *PySMT* [153], which provide highly expressive modeling capabilities and efficient solving algorithms. However, most solvers aim to find any solution, rather than all solutions, as required in the case of auto-tuning search space construction. To obtain all solutions, such solvers must iteratively find a solution, add this solution as an additional constraint, and look for the next solution until there are no solutions left [22]. If there are many solutions, as is commonly the case with auto-tuning problems, this can have a substantial negative impact on performance, as will be shown in Section 4.5.2. A notable exception to this is *python-constraint* [112], as this is a CSP-based Python package capable of finding all solutions, which we will use as the basis of our implemented method.

4.4.2 Parsing Constraints

Having formalized the auto-tuning search space construction to a CSP problem, we must now transform the Kernel Tuner constraints, such as in Listing 2, to a format optimal for CSP solvers. To this end, we introduce a parser for constraints, which has three important benefits: to break down constraints into the smallest subsets of variables, to apply the more efficient specific constraints instead of generic functions where possible, and to provide optimal performance without requiring users of the auto-tuner to write constraints in a complex format that requires an understanding of how CSP solvers work.

To address the latter benefit first, both CSP-solvers and auto-tuners have distinct interfaces consisting of specific function calls or a form of domain-specific language when defining the constraints. As seen before for auto-tuners in Listing 2, an example of a *python-constraint* problem definition is given in Listing 3.

However, as opposed to the users of CSP-solvers, auto-tuning users are generally not aware of the inner workings of the search space construction process and the specific constraints available that result in efficient resolution of the search space. Instead, we provide users the option to write their constraints as Python lambdas or the Python-evaluable string format, both seen in Listing 2, which can then be optimized by our parser via Abstract Syntax Trees. This design has several benefits. It provides a familiar format for constraints, since Python is already used as the interface language. At the same time, our parser can rewrite these constraints, allowing the application of specific constraints instead of generic functions and the decomposition of constraints into subsets.

```
p = Problem()
p.addVariables("block_size_x", [1,2,4,8,16]+[32*i for i in range(1,33)])
p.addVariables("block_size_y", [2**i for i in range(6)])
p.addConstraint(MinProd(32, ["block_size_x", "block_size_y"]))
p.addConstraint(MaxProd(1024, ["block_size_x", "block_size_y"]))
```

Listing 3: Example of a *python-constraint* problem definition.

The automatic reduction of constraints is particularly important to obtain efficiency in practice, as users unfamiliar with the intrinsics of constraint solving, such as the users of auto-tuning frameworks, might write sub-optimal constraints. For example, consider the constraint `2 <= block_size_y <= 32 <= block_size_x * block_size_y <= 1024`, slightly different from Listings 2 and 3, where `block_size_x`, `block_size_y` are tunable parameters with numerical values. Constraints cannot be evaluated until values for the involved parameters are at least partially resolved, resulting in subpar performance in the case of compound statements like the given example, as it depends on the resolution of both parameters. This can be improved by automatically breaking down the composite constraint into multiple constraints with fewer variables where possible. This example can be decomposed as shown in Step 2 of Fig. 4.1, which allows partially resolved values for `block_size_x` or `block_size_y` to discard invalid configurations early.

In addition, this automatic reduction enables the application of specific constraints, as is the case with the example, which can be represented with specific constraints as shown in Step 3 of Fig. 4.1. As will be further discussed in Section 4.4.3, the application of specific constraints can preemptively exclude values through preprocessing, resulting in an even more efficient construction.

4.4.3 Implementation of Optimizations

To obtain the level of performance required to construct auto-tuning search spaces efficiently, we implement several key improvements in various areas of the *python-constraint*

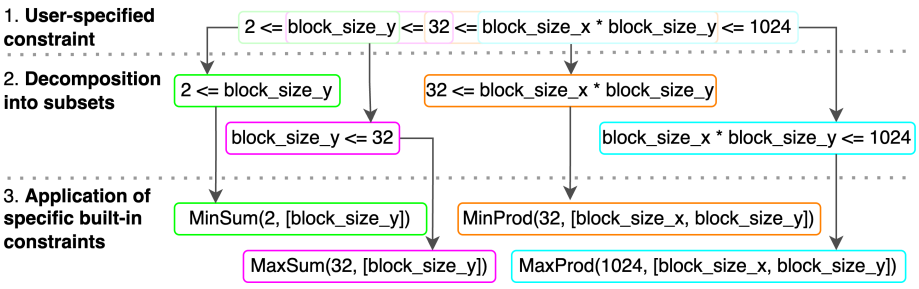


Figure 4.1: The optimization of a constraint via the parsing pipeline.

Algorithm 1: Obtaining all valid configurations as $\mathscr{S}$ (simplified).

Input: Variables $X = \{x_1, \ldots, x_n\}$ with domains D, constraint set C

```
𝒜 ← ∅; 𝒮 ← ∅ ;            ▷ current (partial) assignment; solution set
π ← SortVariables(X) ;      ▷ sorted on number of constraints involved
B ← ⟨(π, 𝒜)⟩ ;                                  ▷ backtrack stack of states
while B ≠ ∅ do                                                // main search loop
    (π, 𝒜) ← B.pop();
    if ∀x ∈ π : x ∈ 𝒜 then // add to solutions if all variables assigned
        𝒮 ← 𝒮 ∪ {𝒜}; continue;
    x ← NextUnassigned(π, 𝒜);
    foreach v ∈ D(x) do                                  // try all values for x
        𝒜[x] ← v;
        if CheckConstraints(𝒜, C) then B.push(π, 𝒜) ;
        𝒜.erase(x) ;                                    ▷ undo and try next value
return 𝒮;
```

package: algorithmically (Section 4.4.3), by extending constraints (Section 4.4.3), engineering (Section 4.4.3), and by tailoring output formats (Section 4.4.3).

We select and optimize a backtracking solver for finding all solutions rather than any solution. Shown simplified in Algorithm 1, it maintains a dictionary of variable assignments and uses a stack-based approach for iterative backtracking, avoiding recursive function calls. For each selected variable, domain values are checked against the constraints. If a constraint is violated, the algorithm backtracks by restoring states until all possibilities are explored. We optimize this algorithm further by sorting the variables on the number of internal constraints, making it faster to find unassigned variables, and by reducing the number of sorts required.

We expand and improve built-in specific constraints to optimize constraint operators commonly used in auto-tuning. By applying knowledge of the operation, their efficiency can be improved over generic functions. For example, given a constraint where $p \cdot q > 0$, we know to ignore all cases where $(p \leq 0) \veebar (q \leq 0)$. We add *MaxProduct* and *MinProduct* constraints as they are commonly used in auto-tuning constraints (e.g. a maximum product of block sizes). We also improve and add preprocessing steps to the various existing constraints, such as *MaxSum* and *MinSum*. All specific constraints are precompiled for further efficiency gains.

However, not all constraints can be expressed as built-in constraints, for example when using an operation that is not as common. Such cases are parsed to *Function* constraints,

which we have optimized by employing function rewriting and dynamic runtime compilation, as the one-off expense of compilation to bytecode is offset by the many times a *Function* constraint is usually executed.

In general, C and similar languages outperform Python in terms of execution speed [180, 2]. To attain this level of performance without losing the flexibility and user-friendliness of Python [5], we employ C-extensions. We transpile the codebase from Python to C-code using Cython [16], which is then compiled into Python-importable C-extensions. We added type hints where possible to aid in compilation.

We implement various output formats to avoid expensive rearrangements to different formats. Expensive rearrangement of the structure in which solutions are output by the solver is mitigated by providing output formats that are close to the internal representation, further described in Section 4.4.4.

4.4.4 Search Space Representation

With the efficient construction of search spaces implemented in *python-constraint*, we consider how this is represented and applied in auto-tuning frameworks for a comprehensive approach.

After the search space construction, optimization algorithms use the information obtained in the construction step to select configurations. Instances of this are obtaining the true bounds of the search space to use balanced initial sampling methods or the selection of valid neighbors that have not been evaluated yet.

This highlights a key advantage of our method over the dynamic approaches discussed in Section 4.3. Important search space characteristics, such as the true parameter bounds, can guide optimization algorithms more effectively and facilitate the use of stratified sampling techniques, such as Latin Hypercube Sampling in Chapter 2. However, these characteristics can not be reliably used in dynamic approaches, as a resolved search space is required. Furthermore, randomized sampling is inherently biased to the sparser parts of the chain-of-trees, although this has been addressed by BaCO [68]. Moreover, selecting valid neighbors of configurations as extensively used by various optimization algorithms is potentially expensive.

Instead, we fully resolve the search space before starting the tuning process, with a minimal impact on the total execution time, to incorporate the full information of the search space in the initial sampling and optimization algorithms. As operations such as

sampling and finding valid neighbors are commonly used in auto-tuning, it can be useful to provide an abstract representation of the search space that implements these operations, providing various views and mappings on the configurations in the search space.

We have implemented this in Kernel Tuner as the *SearchSpace* class, which takes the tunable parameters and constraints based on the user specification, constructs the search space using our implementation, and provides various representations and operations on the resulting search space. The *SearchSpace* class has multiple internal representations for varying purposes, such as hash- and index-based for efficient lookups. Externally, it provides a single interface for all search space-related operations, promoting reuse. For example, the mutation step in the *genetic algorithms* optimization algorithm requires selecting only valid neighbors within a certain Hamming distance. This, along with other neighbor selection algorithms, is implemented in the *SearchSpace* class and can be indexed before running the algorithm, improving overall performance.

4.5 Evaluation

In this section, we evaluate the advancements presented in Section 4.4 to determine their scalability and performance impact. First, we discuss how we compare against the current state-of-the-art solvers in Section 4.5.1. We then evaluate the solvers on a large collection of synthetically generated search spaces with varying characteristics to assess scalability differences in Section 4.5.2. Following this, we evaluate the solvers on a variety of real-world applications to assess the performance improvement in Section 4.5.3. Finally, we validate the practical impact of our method on the entire auto-tuning pipeline in Section 4.5.4.

The evaluations in this work are performed on the sixth generation DAS VU-cluster [8] using an NVIDIA A100 GPU node. The GPU is paired with a 24-core AMD EPYC-2 7402P CPU, 128 GB of memory, and running Rocky Linux 8 with Linux kernel version 4.18. For all tests, the results of each solver were validated against a brute-force solution of the search space. Our evaluation implementation is publicly available.

4.5.1 Comparison against state-of-the-art

To provide additional reference on the performance in this evaluation, we compare the results to the state-of-the-art in auto-tuning search space construction, the chain-of-trees of Auto-Tuning Framework (ATF). ATF has two independent implementations, in C++ [125] and Python [135], both of which we use in this evaluation to compare our method. The C++ version available as of August 2024 with Python bindings is used and denoted as *ATF*

in the results. The Python version called *pyATF* is used at version 0.0.9, the latest version at the time of writing.

Due to the large number of search spaces used in this evaluation, it is not feasible to write each of these search space definition files by hand for both ATF implementations, and we have instead written parsers that define the ATF search space files from an abstract definition of the search spaces. Both implementations of ATF have a notation that combines the definition of tunable parameters, values, and constraints into one statement, as seen in Listing 2. Hence, ATF constraints can only reference tunable parameters that have been previously defined. To provide search space definitions that are compliant with this ATF format, the parsers account for the parameter-constraint order relation and convert to built-in ATF types, such as intervals, where applicable. To reflect the user experience as accurately as possible, the search space file compilation time is included in the total construction time. The C++ version of ATF and search space files is compiled with GCC 12.4.0 using the optimization commands recommended by the ATF documentation.

In addition, we compare with PySMT version 0.9.6, using the Z3 solver developed by Microsoft for software verification and analysis [40]. This allows for evaluation of scalability differences for solvers that do not support enumerating all solutions, as discussed in Section 4.4.1. As with ATF, we developed a custom parser that employs PySMT-specific operations where applicable.

4.5.2 Synthetic Tests

To understand how search space characteristics influence construction time and scaling of solvers, we evaluate on synthetic tests.

We generate a set of search spaces with a varying number of dimensions (between 2 and 5), target Cartesian sizes (with $\{1 \times 10^4, 2 \times 10^4, 5 \times 10^4, 1 \times 10^5, 2 \times 10^5, 5 \times 10^5, 1 \times 10^6\}$), and number of constraints (between 1 and 6). While these arbitrary parameters result in search spaces that are not as large and do not have as many tunable parameters as the real-world search spaces evaluated on in Section 4.5.3, the goal of these in total 78 synthetic search spaces is to gain insight into which solver provides good scalability across the variations in these factors.

Given a Cartesian size, a number of dimensions, and a number of constraints, we want to generate a synthetic search space. To prevent an unfair advantage to solvers optimized for a limited number of dominant dimensions, the number of values per dimension v is kept approximately uniform. This is done by first determining the number of values per dimension as $v = s^{\frac{1}{d}}$, where s is the desired Cartesian size and d is the desired number

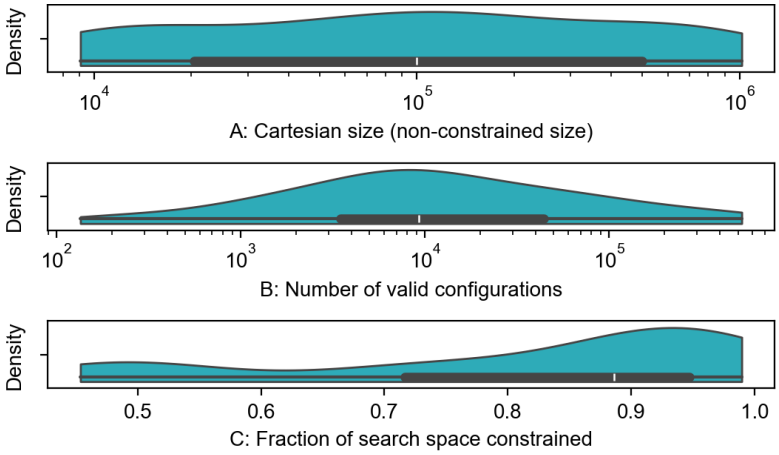


Figure 4.2: Density of three characteristics of the 78 synthetic search spaces. Black bottom bar marks the interquartile range and the white line the median.

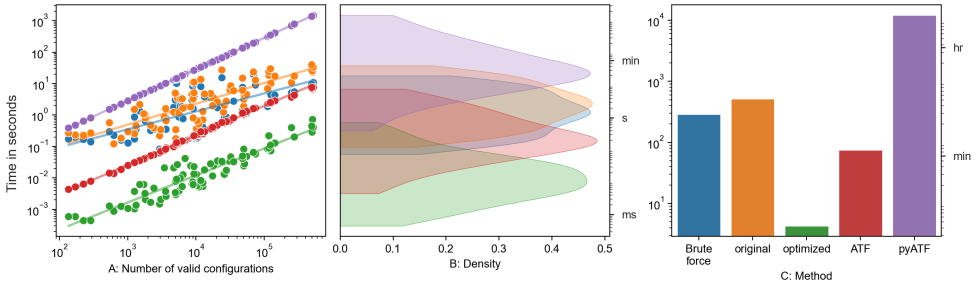


Figure 4.3: Search space construction performance on synthetic tests. Lower times are better. Colors correspond to Figure 4.3C barplot methods. Each plot provides a different view of the same data, with A and B showing the performance on individual search spaces, and C showing the sum of all search spaces.

of dimensions. For each of the dimensions, a linear space with v number of elements is instantiated. Given a non-integer value of v, this is rounded to an integer for all but the last dimension, where v is rounded contradictory (e.g. $5.8 \rightarrow 5$, $5.2 \rightarrow 6$) to be closer to the desired Cartesian size. A list of constraints involving a variety of operations is generated for each combination of dimensions, which are randomly chosen up to the desired number of constraints.

Figure 4.2 shows the distribution of the resulting 78 search spaces for three characteristics. Figure 4.2A shows the actual Cartesian size, representing the total number of possible configurations before constraints are applied, which corresponds to the set of target values. Figure 4.2B depicts the number of valid configurations remaining after constraints are enforced, resulting in an approximately bell-shaped curve. The number of valid configurations is on average one order of magnitude below the Cartesian size. Finally, Figure 4.2C

displays the fraction of sparsity of the search space, i.e., the fraction of non-valid configurations relative to the Cartesian size. Though the fraction of constrained configurations is skewed toward higher values, indicating a propensity towards sparsity, a wide range of variations in sparsity is present.

The performance of the evaluated methods on these synthetic search spaces is displayed in various plots in Figure 4.3, where the colors used correspond to the colors of the methods in the Fig. 4.3C barplot. To determine the impact of the optimizations described in Section 4.4.3, the *original* method denotes the use of vanilla *python-constraint* before the optimizations, whereas our *optimized* method includes the optimizations of Section 4.4.3.

Figure 4.3A shows a clear positive correlation between the number of valid configurations and the execution time across all methods, with a roughly linear trend on the log-log scale, suggesting a power-law relationship. We overlay a log-log linear regression to further investigate the scaling of each method, where a lower slope indicates better scaling towards larger search spaces in the number of valid configurations, and a slope of 1 indicates linear scaling. All methods have a highly significant linear fit with a p-value $\ll 0.05$.

For the methods *ATF* and *pyATF*, we observe approximately linear scaling, with slopes of 0.938 and 0.999, respectively. The *original* unoptimized python-constraint and *brute force* methods appear to perform similarly to each other and exhibit good scaling with slopes of 0.663 and 0.571, respectively. While they are outperformed by *ATF* on these search spaces, the difference in scaling means both methods appear to soon outperform *ATF* on larger search spaces, which we extrapolate to be at $\sim 1.193 \cdot 10^6$ and $\sim 4.493 \cdot 10^7$ valid configurations respectively. It is important to note that this difference in scaling is expected, given that brute-forcing will perform relatively better the denser a search space is (i.e. many valid configurations relative to the Cartesian size of the search space) as it will check the constraints on all configurations, where the chain-of-trees is optimized towards sparse search spaces. Our *optimized* method is consistently the fastest, always constructing the search space within less than a second, and exhibits adequate scaling with a slope of 0.860. As based on this data our *optimized* method would not be overtaken by the *brute force* and *original* methods until $\sim 1.120 \cdot 10^{11}$ and $\sim 3.892 \cdot 10^{15}$ valid configurations respectively, well beyond the size of these synthetic search spaces, we expect our method to perform best overall.

Figure 4.3B presents the performance as a continuous probability density curve using a kernel density estimate (KDE). As expected due to their practically linear scaling, ATF and pyATF demonstrate wide variance in performance. Our *optimized* solver consistently

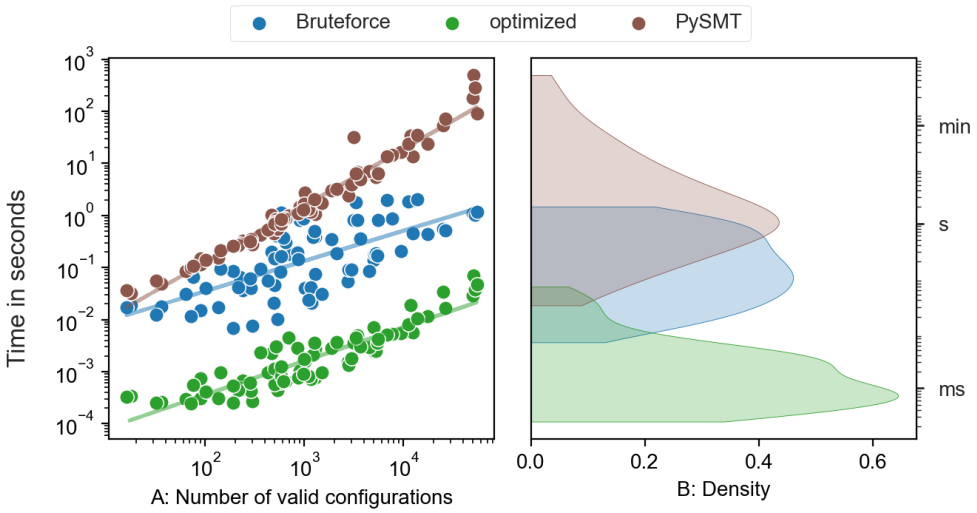


Figure 4.4: Search space construction performance of PySMT on synthetic tests.

achieves the lowest execution times, with several orders of magnitude better performance compared to the other solvers.

Figure 4.3C summarizes the performance of each solver over all search spaces. It is remarkable that *pyATF* takes considerably longer than the *brute-force* method on these search spaces, which might be due to how optimized the chain-of-trees approach is to highly sparse search spaces. It is also noteworthy that our *optimized* method outperforms the *original* unoptimized implementation of *python-constraint* by several orders of magnitude, demonstrating the advantage of our optimizations. Our *optimized* method achieves a 96x speedup over the *brute-force* method (4.75 seconds versus 455.3 seconds), a 16x speedup over *ATF*, and a 2547x speedup over *pyATF*.

As described in Section 4.4.1, a traditional solver without support for finding all solutions requires adding the previous solution as a constraint and iterating over the solutions until all solutions have been found. To demonstrate the lack of scalability of such a solver, Fig. 4.4 compares *PySMT* using the Microsoft Z3 solver to the *brute-force* method and our *optimized* method. To make executing this experiment feasible, we reduce the size of the generated synthetic search spaces by one order of magnitude in this experiment. As seen in Fig. 4.4, PySMT performs poorly relative to both brute force and our optimized method. As expected, this difference increases as the number of valid configurations increases, demonstrating the infeasibility of this approach when many valid configurations are present. Despite the reduced search space sizes, PySMT with the Z3 solver still takes nearly a thousand seconds on the largest search spaces, whereas the brute-force solver takes about ten seconds. Our optimized solver vastly outperforms PySMT, taking about

Table 4.2: Overview of the basic characteristics of the real-world search spaces and the mean values for each of the columns.

Name	Cartesian size	Constraint size	Number of parameters (dimensions)	Number of constraints	Avg. unique parameters per constraints	Range of number of values per parameter	% of configurations in Cartesian size	Avg. number of constraint evaluations required
Dedispersion	22272	11130	8	3	2	1 - 29	49.973	33414
ExpDist	9732096	294000	10	4	2	1 - 11	3.021	23889240
Hotspot	22200000	349853	11	5	3.8	1 - 37	1.576	65900294
GEMM	663552	116928	17	8	3.25	1 - 4	17.622	2576736
MicroHH	1166400	138600	13	8	2.375	1 - 10	11.883	4763700
ATF PRL 2x2	36864	1200	20	14	2.429	1 - 3	3.255	268680
ATF PRL 4x4	9437184	10800	20	14	2.429	1 - 4	0.114	70708680
ATF PRL 8x8	2415919104	48720	20	14	2.429	1 - 8	0.002	18119076600
Mean	*307322534*	*121403*	*14.875*	*8.75*	*2.589*	*1 - 13.25*	*10.93*	*2285902168*

as long to solve the largest search spaces as PySMT with the Z3 solver takes to solve the smallest search spaces. In fact, the PySMT solver exhibits superlinear scaling with a slope of 1.090, as opposed to the slope of 0.649 of our optimized method. As it is infeasible to evaluate the large search spaces of the selected real-world applications, PySMT with the Z3 solver will not be included in the remainder of the evaluation.

4.5.3 Real-world Applications

To evaluate solver performance on the search spaces of real-world applications, we select the three largest search spaces in the Benchmark suite for Auto-Tuners (BAT) [159]. These are *Dedispersion*, *Hotspot*, and *ExpDist*. In addition, we use the relatively large search spaces of the *MicroHH* computational fluid dynamics kernel [160], as well as the commonly used General Matrix Multiplication kernel *(GEMM)* [115]. To provide reference points for a fair comparison to ATF, the Probabilistic Record Linkage (*PRL*) kernel used in the chain-of-trees evaluation [126] is used as well, resulting in three additional search spaces for a total of eight real-world search spaces.

The characteristics of the real-world search spaces are displayed in Table 4.2, where the rightmost column denotes the average number of constraint evaluations that are required to brute-force solve a search space. For each combination in the Cartesian product, all constraints need to be evaluated until the combination is rejected or all constraints have been evaluated. Hence, assuming uniform probability among the constraints, the average number of constraint evaluations can be calculated by taking the average of the best case (the first constraint rejects the combination) and worst case (the last constraint rejects the combination), and adding all valid combinations that are never rejected. Given a search space S, let S_i be the set of non-valid combinations, S_v the set of valid combinations, and

S_c the set of constraints, the average number of constraint evaluations can be calculated as $\frac{|S_i|+|S_i|\cdot|S_c|}{2} + |S_v|$. Descriptions of each of the kernels and their search spaces are given in Section 4.5.3, before the results are discussed in Section 4.5.3.

The Dedispersion kernel introduced in [136, 137] and used in [159] is designed to compensate for the time delay experienced by radio waves as they propagate through space. This delay occurs due to the frequency-dependent dispersion of the signal. By applying a specific dispersion measure (DM) and reversing the dispersion effect, the kernel reconstructs the original signal. During the iteration over frequency channels, threads process multiple time samples and dispersion measures in parallel. Comparing the Dedispersion search space to the other evaluated search spaces of Table 4.2, the resulting search space is the smallest in Cartesian size, but as it has the highest percentage of valid configurations at nearly 50 %, it is not the smallest in number of valid configurations.

The ExpDist kernel described in [159] is utilized in a localization microscopy application that performs template-free particle fusion by integrating multiple observations into a single super-resolution reconstruction [69]. During the registration process, the ExpDist kernel is repeatedly invoked to evaluate the alignment of two particles. The algorithm exhibits quadratic complexity with respect to the number of localizations per particle, making it highly computationally intensive. The resulting search space is the second-most sparse of the real-world search spaces in Table 4.2.

Previously discussed in Section 4.2, the Hotspot kernel of [159] is part of a thermal simulation application to estimate the temperature of a processor by considering its architecture and simulating power currents. Through an iterative process, the kernel solves a set of differential equations. The inputs to the kernel consist of power and initial temperature values, while the output is a grid displaying average temperature values across the chip. It is interesting to note that the Hotspot search space is the largest in number of valid configurations, second-largest in Cartesian size, and has the highest number of values for a single parameter.

The computational fluid dynamics kernel of [160] is used for weather and climate modeling, specifically for the simulation of turbulent flows in the atmospheric boundary layer.

In this case, we use the search space resulting from the auto-tunable GPU implementation of the `advec_u` kernel with extended parameter values as specified in the source of [67]. Looking at Table 4.2, it is notable that the MicroHH search space is the closest to the mean values of all search spaces in the number of parameters, number of constraints, and percentage of configurations. It is also second-closest in constraint size and number of values per parameter, making it perhaps the most average search space in our set of tests.

Generalized dense matrix-matrix multiplication is a fundamental operation in the BLAS linear algebra library and widely used across various application domains. Known for its high performance on GPU hardware, GEMM frequently serves as a benchmark in studies of GPU code optimization [114, 90, 133]. In this evaluation, we use the GEMM kernel of CLBlast [115], a tunable OpenCL BLAS library. GEMM is implemented as the multiplication of two matrices (A and B); $C = \alpha A \cdot B + \beta C$, where α and β are constants and C is the output matrix. The dimensions of all three matrices are set to 4096×4096, resulting in a dense search space.

The Probabilistic Record Linkage (PRL) kernel used in [126] is a parallelized implementation of an algorithm that is commonly used in data mining to identify data records referring to the same real-world entity. In this kernel, the input sizes determine the size of the search space. As shown in Table 4.2, the brute-force resolution of this search space with input sizes $8x8$ requires 1.8119×10^{10} constraint evaluations on average, which took $\sim$27 hours to execute. As an input size of $16x16$ would require 4.639×10^{12} constraint evaluations on average, it is not feasible to brute force beyond the $8x8$ input size. Because the brute-forced solution is used for validation and serves as a reference point in the performance comparisons, we use the search spaces resulting from the ATF PRL kernel with input sizes $2x2$, $4x4$, and $8x8$. It is notable that while the $8x8$ search space results in the largest Cartesian size of the set, the ATF PRL search spaces are very sparse.

Figure 4.5 presents the search space construction performance across the eight real-world benchmarks for the five different constraint solver methods, as before in Section 4.5.2.

Figure 4.5A and Fig. 4.5B illustrate the relationship between search space size and solver performance, with a log-log linear regression overlayed where significant (p-value ≤ 0.05), as in Section 4.5.2. In general, larger constrained search spaces (A) and Cartesian sizes (B) result in increased search times, as previously observed in Fig. 4.3. For the

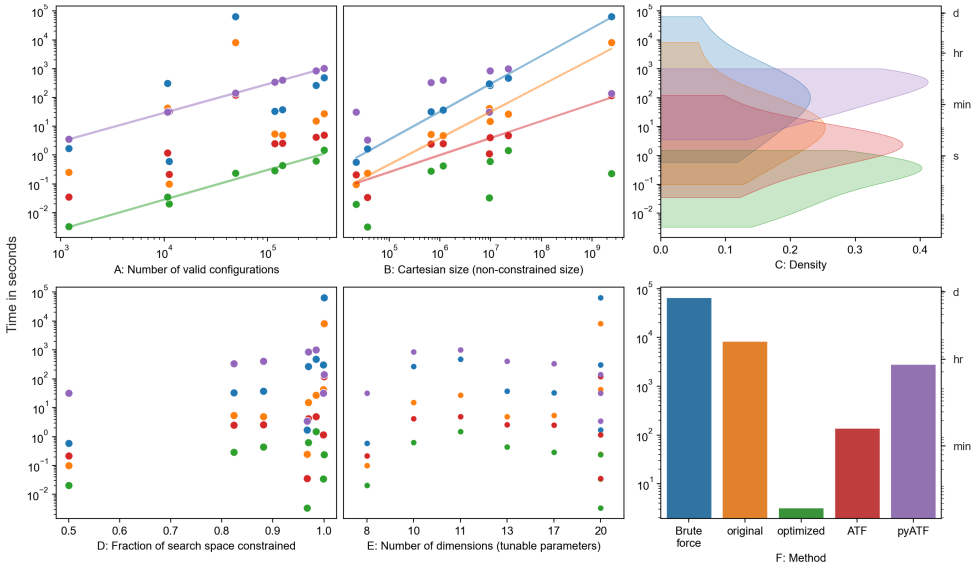


Figure 4.5: Search space construction performance on real-world tests. Lower times are better. Colors correspond to Figure 4.5F barplot methods. Each plot provides a different view of the same data; plots A-E show individual performance relative to a search space characteristic, and plot F shows the sum of all search spaces.

ATF, *original*, and *brute-force* methods, the significant scaling trend is along the Cartesian size of Fig. 4.5B, whereas for *pyATF* and our *optimized* method, this is on the number of valid configurations in Fig. 4.5A. Our *optimized* method achieves the lowest execution times across all search space sizes, demonstrating its efficiency, and is the only solver that consistently outperforms the other methods.

Figure 4.5C visualizes the distribution of execution times, providing an indication of the average performance and variability. It is interesting to observe that while the *original* python-constraint method is one order of magnitude faster than the *brute-force* method, both methods have very similar distributions, as seen before in Fig. 4.3B. A clear trend emerges from this plot, where our *optimized* method has the least variability and is the only solver constructing the search spaces in the sub-second domain.

In Fig. 4.5D, the relation between how constrained a search space is and solver performance is displayed. *ATF* and *pyATF* performance appears to be strongly influenced by the sparsity, as for fraction > 0.9 *ATF* performance is substantially better than the *original* solver, in contrast to ≤ 0.9, where at fraction $\simeq 0.5$ even the unoptimized *original* python-constraint outperforms *ATF*.

The number of tunable parameters displayed in Fig. 4.5E does not appear to have as much of an impact on performance as the other plots discussed. Nevertheless, Fig. 4.5E is

useful to discern the individual search spaces based on the number of parameters described in Table 4.2. For instance, it can be noted that the performance difference between our optimized method and all other methods appears to be relatively stable, even for the ATF PRL search spaces, as can be discerned by the number of tunable parameters, where the three ATF search spaces have 20 tunable parameters.

Finally, Fig. 4.5F summarizes the total time taken by each solver. The brute-force approach is the least performant, taking nearly a full day to resolve the eight search spaces. Although the *original* python-constraint solver is faster than brute force, our *optimized* solver achieves a ∼2643x speedup over it, demonstrating the efficiency of our optimizations. While ATF and pyATF achieve intermediate performance levels, it must be noted that pyATF only outperforms *brute force* and *original* because it does so on the two largest PRL search spaces, which have a disproportionate effect on the summed time - on all other search spaces, pyATF is outperformed by both methods, and ATF is not consistently better than *original* either. The optimized solver consistently and considerably outperforms all others: our *optimized* method achieves a ∼20643x overall speedup over the brute-force method (3.16 seconds versus 65230.47 seconds), ∼44x over ATF, and ∼891x over pyATF.

4.5.4 Overall impact in practice

To conclude this evaluation, we evaluate the impact of the search space construction method on the overall auto-tuning process.

We auto-tune the *hotspot* kernel described in Section 4.5.3 using the three Python-based solvers with a 30-minute time budget as an illustrative example. To avoid influence by a specific optimization algorithm, we use random sampling, and each run is repeated 10 times. Figure 4.6 shows the best-performing configuration found so far during the tuning process, where higher is better. The time passed before any configuration is found is spent constructing the search space, which takes about eight minutes for brute-force and takes over twenty minutes for pyATF, while our optimized method is able to start to tune configurations almost immediately.

To affirm these findings, we repeat this experiment on the *GEMM* kernel, adjusting the time budget by the ratio between the number of valid configurations of GEMM and *hotspot* seen in Table 4.2 to 10 minutes. While brute force fares substantially better, as expected due to the smaller and denser search space, the results are otherwise very similar to those of the previous experiment. Most importantly, both examples confirm that the search space construction method has a substantial impact on the quality of the overall best configuration found within the time budget.

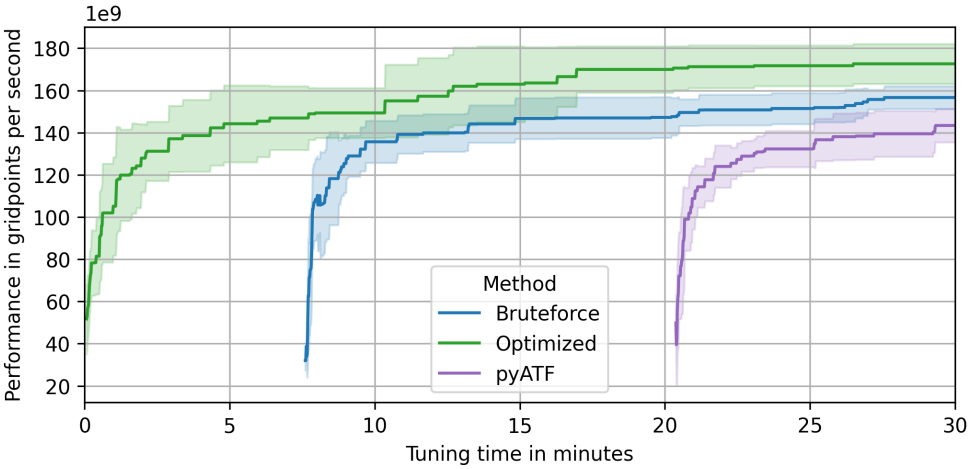


Figure 4.6: Best configuration performance found over a 30-minute auto-tuning of the *hotspot* kernel using various search space construction methods.

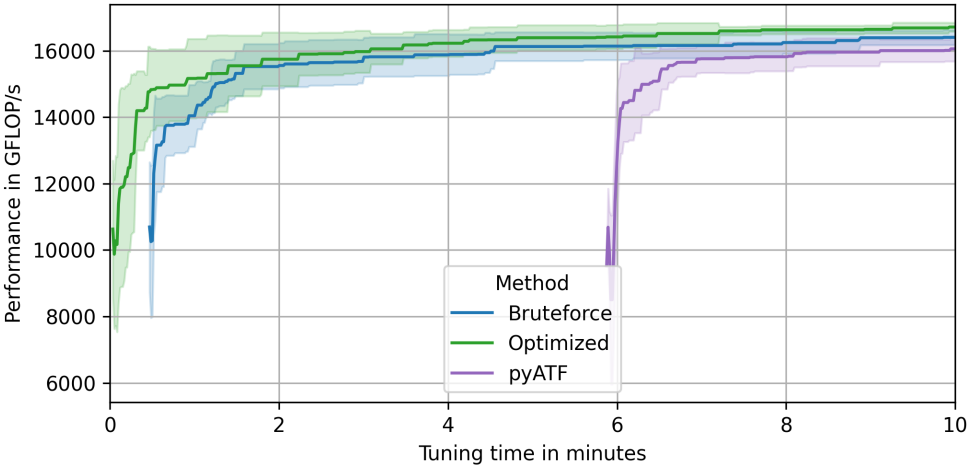


Figure 4.7: Best configuration performance found over a 10-minute auto-tuning of the *GEMM* kernel using various search space construction methods.

Overall, throughout this evaluation section, it is noteworthy that our optimized solver consistently outperforms any alternative on all of the search spaces by a wide margin, and the substantial practical impact this can have on the overall auto-tuning process. These findings emphasize the advantages of our optimized solver in efficiently handling large and complex search spaces.

4.6 Conclusions

We introduced a novel approach to constructing auto-tuning search spaces using an optimized Constraint Satisfaction Problem (CSP) solver, addressing the specific challenges posed by the complexity of auto-tuning and large search spaces. Our contributions, available to the CSP-solving and auto-tuning community in the open-source python-constraint2 and Kernel Tuner packages, substantially outperform state-of-the-art methods in search space construction performance, enabling the exploration of previously unattainable problem scales in constraint-based auto-tuning and related domains.

Through rigorous evaluation, we demonstrated that our optimized CSP-based approach reduces construction time by several orders of magnitude, even for search spaces with billions of possible combinations. On average over the evaluated real-world applications, our optimized method is four orders of magnitude faster than brute force, three orders of magnitude faster than the unoptimized CSP solver, and one to two orders of magnitude faster than the state-of-the-art in search space construction. Our optimized search space construction method reduces the construction time of real-world applications to sub-second levels, eliminating it as a substantial factor in the overall tuning process overhead. In addition, our parsing method allows users to write constraints that are as close to the target language as possible, improving accessibility. Furthermore, our method prevents skewed sampling and has additional benefits for the efficiency of auto-tuning optimization algorithms. This breakthrough allows researchers and developers to more effectively harness the performance potential of modern hardware and provides an efficient generic solver for similar problem domains.

5 Tuning the Tuner: Introducing Hyperparameter Optimization

> *Give me six hours to chop down a tree,*
> *and I will spend the first four sharpening the axe.*
>
> attributed to Abraham Lincoln

Optimization algorithms in auto-tuners have hyperparameters that strongly influence performance, yet these are rarely tuned in practice. We introduce a general method for hyperparameter optimization of auto-tuning algorithms, including a statistical evaluation procedure, a public dataset, and a simulation mode to reduce tuning costs by two orders of magnitude. Experiments show that even limited hyperparameter tuning can yield average performance gains of 94.8%, with meta-optimization providing gains over 200%. These results establish hyperparameter tuning as an impactful tool for advancing auto-tuning efficiency.

This chapter is based on "Tuning the Tuner: Introducing Hyperparameter Optimization for Auto-Tuning" by Willemsen, van Nieuwpoort, and van Werkhoven as published in [172].

5.1 Introduction

At the heart of the auto-tuning method is a search space of functionally-equivalent *code variants* that is explored by an optimization algorithm. Together, these code variants constitute vast search spaces that are infeasible to search by hand [133, 136, 167] and would have to be searched over and over again as the application is executed on different hardware or different input data sets and sizes [115, 166, 84, 121]. Therefore, a key problem in auto-tuning research is the development of optimization algorithms that can efficiently navigate the auto-tuning search space [165, 134].

The effectiveness of optimization algorithms is strongly influenced by their *hyperparameters*, i.e., settings that control how the algorithm explores the search space. Hyperparameter tuning is essential in fields such as machine learning to determine optimal algorithm settings [12], e.g., learning rates [77], neural architectures [88], and regularization parameters [18]. While hyperparameter tuning has been successfully applied in a wide variety of areas, it has to the best of our knowledge not been applied to optimization algorithms for auto-tuning. For example, the surveys on auto-tuning in High-Performance Computing by Balaprakash et al. [9], as well as the comprehensive survey by Ashouri et al. [4] on machine learning in auto-tuning do not mention hyperparameter tuning at all. As such, the potential impact of hyperparameter tuning on the performance of optimization algorithms for auto-tuning has not been quantified or studied, and no generic auto-tuning framework currently supports it.

To bridge this gap and enable hyperparameter tuning for the field of auto-tuning, we need to overcome two key challenges:

1. The hyperparameter tuner must optimize performance across different auto-tuning search spaces to achieve good generalization across different applications and hardware architectures. However, quantifying overall performance across different applications and architectures using different performance metrics is nontrivial.
2. To achieve robust general performance estimates of stochastic optimization algorithms currently necessitates a high number of repeated tuning runs, which makes hyperparameter tuning prohibitively expensive.

In this work, we address these challenges and introduce a novel approach to hyperparameter tuning tailored to the problem of auto-tuning that we refer to as ”tuning the tuner”. We make the following contributions:

- We propose a systematic and general method for hyperparameter tuning of optimization algorithms for auto-tuning.
- We propose a novel *simulation mode* that accelerates hyperparameter tuning by simulating optimization runs, yielding a $\sim 130\times$ speedup.

- We provide a FAIR dataset of general auto-tuning data that is well-balanced in applications and hardware, lowering barriers to entry and reducing energy and resource costs.
- We present the first evaluation of the impact of hyperparameter tuning on the performance of optimization algorithms for auto-tuning, with an average performance improvement of 94.8% on a limited, exhaustively evaluated set of hyperparameters.
- We present the first results on the effectiveness of meta-strategies for efficient hyperparameter tuning for auto-tuning, enabling application beyond exhaustive tuning, with an average improvement of 204.7% on an extended set of hyperparameters tuned with a meta-strategy.

All contributions are implemented in the open-source Kernel Tuner [165] and Autotuning Methodology frameworks (described in Chapter 3).

Given the possible confusion in terminology in distinguishing between auto-tuning and hyperparameter tuning, we will use *kernel configuration* to refer to the specific settings of a to-be-tuned computational kernel, and *hyperparameter configuration* for the specific settings of an optimization algorithm. The rest of this work is structured as follows. Section 5.2 discusses related work, Section 5.3 describes our method and implementation, Section 5.4 evaluates the diverse aspects of our method, and Section 5.5 concludes the chapter.

5.2 Related Work

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. [9]. As stated in the previous section, to the best of our knowledge, the application and impact of hyperparameter tuning in the context of auto-tuning is largely unstudied. As such, this section reviews the support for optimization algorithms and the possibility of controlling their hyperparameters in the current state-of-the-art generic auto-tuners, and reviews a handful of studies that include a preliminary evaluation of hyperparameters in auto-tuning.

Table 5.1 presents an overview of generic auto-tuning frameworks, the optimization algorithms they implement, and their support for setting hyperparameters. As we can see, many of these frameworks implement advanced search strategies, but few offer a practical interface to configure hyperparameters without modifying the tuner itself. For example, OpenTuner [3], KTT [49], ATF [125], PyATF [135], and GPTune [95] do not support specifying the hyperparameters without source code modifications.

Falch and Elster have presented the AUMA [47] auto-tuner to improve the performance portability of OpenCL applications across various parallel architectures, including

Table 5.1: Overview of auto-tuning frameworks, their open source status, supported optimization algorithms, and support for configuring hyperparameters without modifying source code. SA = Simulated Annealing. PSO = Particle Swarm Optimization.

Framework	Open Source	Active	Optimization algorithms	Hyperparameters
OpenTuner [3]	✓	✗	SA, PSO, Multi-armed Bandit, Evolutionary methods, Local Search	✗
KTT [49]	✓	✓	Markov Chain Monte Carlo, Profile-based search	✗
ATF [125] / PyATF [135]	✓	✓	SA, Differential Evolution, Multi-armed Bandit, Local Search	✗
GPTune [95]	✓	✓	Bayesian Optimization	✗
AUMA [47]	✓	✗	K-Bagging Neural Networks	✓(File-based)
BaCO [68]	✓	✗	Bayesian Optimization, Exhaustive	✓(File-based)
ytopt [175]	✓	✓	Bayesian Optimization	✓(command line)
CLTune [114]	✓	✗	SA, PSO, Neural Network	✓(API-based)
Kernel Tuner [165]	✓	✓	20+ global and local optimization algorithms, including Annealing, Evolutionary, and Swarm-based methods, and Bayesian Optimization	✓(API-based)

Intel CPUs and GPUs, as well as AMD and Nvidia GPUs. AUMA uses a machine-learning approach to predict the subset of the search space that is likely to contain high-performant code variants. Their method implements one hyperparameter called the *threshold* that controls when to stop tuning. They evaluate their methods for four different thresholds, showing that the optimal value strongly depends on the tuned application.

The BaCO [68] auto-tuner supports setting the hyperparameters in a JSON file as part of the specification of a tuning problem. However, BaCO is not evaluated or tested with different hyperparameters in the accompanying paper [68].

Nugteren and Codreanu [114] implemented Simulated Annealing (SA), Particle Swarm Optimization (PSO), as well as a neural network-based optimizer, in CLTune. CLTune provides an API-based interface for controlling the hyperparameters, and the authors do a very limited evaluation of hyperparameter sensitivity. They evaluate three different values for the *temperature* hyperparameter of SA, as well as two different values for the *swarm size* of PSO for auto-tuning a 2D convolution kernel on four GPUs. However, their evaluation shows only minimal performance differences, with no optimization algorithm

implemented in CLTune outperforming random search. The authors do note that "the evaluated search strategies are based on stochastic variables, we cannot draw conclusions from this limited set of experiments", which underlines the need for robust hyperparameter tuning methods.

Kernel Tuner [165] is the only actively developed open-source auto-tuner with an API for controlling the hyperparameters. As such, we have chosen Kernel Tuner as the tuner in which we implement our method.

The most closely related work is perhaps the survey of optimization algorithms for auto-tuning by Schoonhoven et al. [134], in which a limited hyperparameter tuning is conducted to compare optimization algorithms for auto-tuning. They measure performance by counting head-to-head wins between two algorithms on individual problems and select hyperparameter tuning configurations by repeatedly combining configurations with the most wins from individual problems. However, generalization is not addressed, and the performance impact of hyperparameter tuning is not evaluated in their study.

5.3 Design and Implementation

In this work, we present a method for general hyperparameter tuning for auto-tuning; this section provides an overview of its design and implementation. Section 5.3.1 provides an introduction of the auto-tuning problem, leading into Section 5.3.2 which introduces the proposed hyperparameter tuning method. Section 5.3.3 presents our simulation mode to make hyperparameter tuning feasible. Section 5.3.4 presents our FAIR data set for benchmarking auto-tuner performance, and Section 5.3.5 discusses the implementation.

5.3.1 The auto-tuning problem

Auto-tuning involves optimizing an application or *kernel* K_i on a target system G_j for input data I_k to maximize performance $f_{G_j,I_k}(K_i)$. The auto-tuner constructs a search space $\mathscr{X}$ by considering all tunable parameters and their valid values, subject to user-defined constraints, as described in Chapter 4. The objective is to determine the optimal configuration, or more formally (assuming minimization) as follows:

$$x^\star = \arg\min_{x \in \mathscr{X}} f_{G_j,I_k}(K_{i,x}). \tag{5.1}$$

The evaluation of each configuration takes time, as it needs to be compiled and executed on the target system. Search spaces in auto-tuning are often large, discontinuous, non-convex, and irregular. These characteristics make them infeasible to search by hand and demand carefully chosen optimization algorithms. If the algorithm is not well adapted

to the search space, the number of required evaluations becomes excessive, limiting the practical usability of the auto-tuner.

5.3.2 Generalized Hyperparameter Tuning

Hyperparameter tuning adjusts the settings of an optimization algorithm to enhance its efficiency across a domain of problems. Examples of hyperparameters include the population size in evolutionary algorithms, neighbor selection in local search methods, and temperature schedules in annealing-based approaches. While auto-tuners typically optimize performance-related parameters for a specific application on a specific target system, our hyperparameter tuning approach aims for *general optimized performance across multiple tuning problems*.

In Chapter 3, we presented a community-driven methodology for evaluating optimization algorithms in auto-tuning, which we extend to hyperparameter tuning. The methodology provides a systematic approach to comparing optimization algorithms across auto-tuning search spaces. It defines a performance score that quantifies an optimization algorithm's performance over the passed time relative to a calculated baseline, typically random search, to have consistent, objective-independent, transparent, and comparable behavior across search spaces.

On an individual search space, this approach first defines a budget and performance baseline adapted to the search space characteristics. The optimization algorithms to be compared are then executed multiple times to account for noise, after which the difference between the baseline and the average best performance of each optimization algorithm is compared at fixed, equidistant time intervals relative to the budget. The result is a smooth performance curve over time, instead of relying solely on final performance. By adapting the budget and baseline to reliable search space characteristics, we can compare and aggregate these performance curves across search spaces. We will thus use the mean of these aggregated performance curves as a performance score $\mathscr{P}$ for general hyperparameter tuning of an optimization algorithm, where a higher score is better overall performance.

More formally, for a given time sampling point t the performance $\mathscr{P}_t$ of an optimization algorithm $\mathscr{F}$ can be computed:

$$\mathscr{P}(\mathscr{F}, G_j, K_i, I_k)_t = \frac{\mathscr{S}_{\text{baseline}}(t) - \mathscr{F}(G_j, K_i, I_k)_t}{S_{\text{baseline}}(t) - \mathscr{S}_{\text{opt}}} \tag{5.2}$$

where $\mathscr{S}(G_j, K_i, I_k)$ are the search space characteristics given a target system G_j, kernel K_i, and input data I_k. Here, $\mathscr{F}_t$ is the best objective value found so far by the optimization algorithm, $\mathscr{S}_{\text{baseline}}(t)$ is the performance of the baseline method, and $\mathscr{S}_{\text{opt}}$ is the

known optimal value in the search space. This yields $\mathscr{P}_t = 0$ when performance equals the baseline and $\mathscr{P}_t = 1$ when the optimum has been found.

To avoid distorting the metric with large variations in search space difficulty, evaluations are limited to the time at which the baseline reaches a set cutoff percentile between the median and the optimum - typically somewhere around 95% - referred to as the budget. With this method, the performance curves are relative to the same baseline and optimum, the cutoff provides a consistent reference point, and the sampling points are equidistant. The performance curves can thus be meaningfully aggregated from different search spaces by taking the mean score of all curves at each t, resulting in the aggregate performance curve over all search spaces for the optimization algorithm. The aggregate performance score is then obtained by averaging over the set of discrete time sampling points $\mathscr{T}$, or more formally:

$$\mathscr{P}(\mathscr{F}, K, G, I) = \frac{1}{|\mathscr{T}|} \sum_{t \in \mathscr{T}} \frac{\sum_{K_i \in K} \sum_{G_j \in G} \sum_{I_k \in I} \mathscr{P}(\mathscr{F}, G_j, K_i, I_k)_t}{|K||G||I|} \tag{5.3}$$

The kernels K, target systems G, and inputs I are the collections of K_i, G_j, and I_k of Eq. (5.1) on which the hyperparameter tuning is conducted, also referred to as the training set. This aggregate score enables robust comparison of optimization algorithms by capturing both the quality of the configurations found as well as the time taken to do so. As such, a difference when comparing two performance scores can indicate a difference in the quality of configurations found, the time taken to do so, or a combination of both.

The hyperparameter tuning problem can thus be formalized:

$$h^\star = \arg\max_{h \in H} \mathscr{P}(\mathscr{F}_h, K, G, I) \tag{5.4}$$

where H represents the hyperparameter space, and $\mathscr{F}_h$ is the optimization algorithm configured with hyperparameters h.

5.3.3 Hyperparameter Tuning Feasibility

It must be noted that evaluating Eq. (5.4) requires that for each hyperparameter configuration h in H, an auto-tuning experiment is run on each combination of the involved kernels, target systems, and input datasets as seen in Eq. (5.3), leading to a combinatorial explosion in computational cost. In addition, this requires that all systems used 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. These factors combined make executing such a hyperparameter tuning prohibitively expensive and time-consuming.

To address this challenge, we introduce a *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 runs. For such a simulation mode to work, all configurations in a search space must have been evaluated on the actual hardware (an exhaustive or *brute-force* auto-tuning search). This might seem counterintuitive, 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 tuning the hyperparameters of 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 target systems. In addition, assume each of the 9 resulting search spaces has 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 5.1, most auto-tuning frameworks provide multiple optimization algorithms.

The simulation mode approach provides several important advantages. First, it substantially improves efficiency: after the high one-off cost of brute-forcing, the threshold of comparison to and hyperparameter tuning of additional optimization algorithms is substantially lower; due to the discrete nature of the search spaces combined with how the budget works (Section 5.3.2) and multiple repeats required for stochastic optimization algorithms, configurations are likely to be revisited. In addition, it mitigates measurement noise: the simulation mode ensures every auto-tuning execution is reproducible, preventing additional noise in the process. Finally, it reduces energy use and resource contention: once the full search space is evaluated, hyperparameter tuning can be performed without costly compilation and execution on the target system, 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 data 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 publicly

Table 5.2: Brute-force execution times in hours for each search space.

Application	A100	A4000	A6000	MI250X	W6600	W7800
Dedispersion	10.2	19.9	12.2	19	21.4	10.4
Convolution	3.4	3.5	3.5	2.9	4.5	2
Hotspot	100.9	32.9	25.5	51	73.1	50.3
GEMM	44.7	58.4	40	61.2	250.6	60.9

available provides and promotes a reproducible record that can be used and contributed to by the community.

5.3.4 FAIR benchmark data for Hyperparameter Tuning

To enable reproducible and scalable hyperparameter tuning of optimization algorithms for GPU auto-tuning in particular, we have developed and curated a dataset of fully brute-forced search spaces across a diverse set of kernels and GPU architectures. This dataset, referred to as the *Benchmark Hub for Auto-Tuning*, serves as a community resource to facilitate benchmarking and collaborative research. It consists of 24 exhaustively evaluated search spaces, formed by the Cartesian product of four real-world GPU kernels across six GPUs. Each kernel configuration in these search spaces has been executed 32 times to mitigate measurement noise, with both the average and raw values present in the data.

The four applications are the *dedispersion*, *convolution*, *hotspot*, and *GEMM* kernels as described by [99], widely used in astronomy, image processing, material science, and linear algebra, respectively. Dedispersion is a signal-processing kernel that reconstructs radio signals distorted by interstellar dispersion by applying a range of dispersion measures to time-domain samples across multiple frequency channels. The 2D convolution stencil kernel performs image filtering by computing weighted sums over image regions. Hotspot is a thermal simulation stencil kernel that estimates processor temperature by iteratively solving differential equations based on simulated power and initial temperature inputs, producing a temperature grid as output. GEMM (General Matrix-Matrix Multiplication) is a linear algebra operation implemented in CLBlast that computes $C = \alpha A \cdot B + \beta C$ for large dense matrices. These applications also bring diversity in their performance characteristics; e.g., dedispersion and hotspot are generally bandwidth-bound, convolution and GEMM are generally compute-bound.

The six GPUs are an AMD MI250X of the LUMI supercomputer, and an AMD W6600, AMD W7800, Nvidia A6000, Nvidia A4000, and Nvidia A100 of the DAS-6 supercomputer [8]. The time taken to execute the exhaustive brute-force search for each search space is listed in Table 5.2, for a total of just over 962 hours to compute the entire dataset.

In alignment with the vision outlined in *FAIR sharing of data in autotuning research* [75], our dataset adheres to the FAIR principles:

- **Findable:** All benchmark data and associated scripts are publicly available through Zenodo[1] and a dedicated GitHub repository[2], with clear structure, documentation, and Digital Object Identifiers (DOIs) for unique versions.
- **Accessible:** The data is open access, licensed for reuse, and includes versioned kernel code, configuration files, and benchmark results.
- **Interoperable:** Input data is stored in the *T1* format, while output data follows the *T4* format, as stipulated in [75]. These standardized JSON-based formats are compatible with various auto-tuning frameworks such as KTT [49], Kernel Tuner [165], as well as the Autotuning Methodology software package.
- **Reusable:** The dataset is directly usable in simulation mode, enabling evaluation of optimization strategies and hyperparameter tuning *without requiring access to the original target systems.* Integration and utility scripts are provided to facilitate reuse and extension.

The repository is structured to support contributions: new kernels or brute-forced search spaces on additional GPUs can be submitted by the community. Each kernel directory includes the original source code, T1 input description, and tuning script. Corresponding output data is provided per GPU in both the original tuner's format and the interoperable T4 format. To optimize storage and portability, output files are compressed and decompressed automatically.

This FAIR-compliant dataset empowers scalable experimentation, enables generalization across a diverse range of applications and GPUs, reduces resource usage, encourages collaboration, and lowers the barrier to entry for reproducible research in auto-tuning.

5.3.5 Integration with Kernel Tuner

To demonstrate the real-world impact of our proposed method, we implement it in an actively developed open-source auto-tuning framework. Kernel Tuner is a Python-based auto-tuner designed to optimize GPU kernels for various objectives, such as execution time and energy efficiency [165]. It supports CUDA, HIP [99], 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

[1] https://doi.org/10.5281/zenodo.15364124
[2] https://github.com/AutoTuningAssociation/benchmark_hub

efficiently [134]. The wide variety of implemented optimization algorithms and provided interface for setting hyperparameter configurations make Kernel Tuner an appropriate candidate for implementing our hyperparameter tuning method.

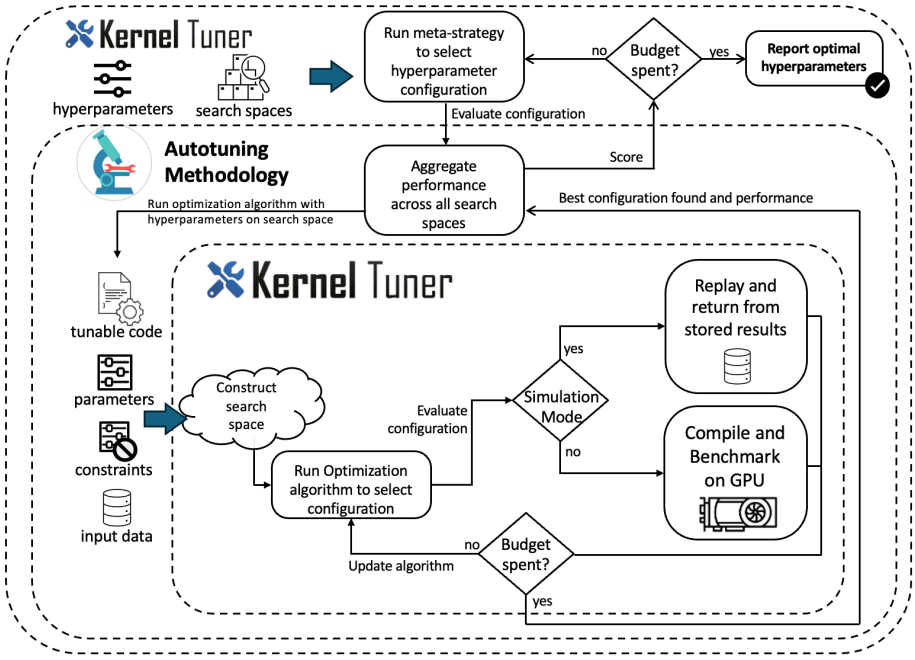


Figure 5.1: The hyperparameter tuning pipeline for auto-tuning. Kernel Tuner's hyperparameter tuning functionality (the outermost layer) calls the autotuning methodology software to get an aggregate performance score, which is obtained by running the optimization algorithm on various search spaces.

An abstract visualization of our hyperparameter tuning pipeline is shown in Fig. 5.1. The hyperparameter tuning we implemented in Kernel Tuner uses the autotuning methodology software package of Chapter 3 to request the performance score $\mathscr{P}$ of Eq. (5.3) for an optimization algorithm with a selected hyperparameter configuration across the search spaces selected for training. 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. 5.1 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, benefiting 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 more informed searches for hyperparameter tuning configurations compared to exhaustive search, and should allow near-optimal hyperparameter configurations to be found more efficiently.

5

5.4 Evaluation

In this section, we evaluate the advancements presented in Section 5.3 to determine the efficacy and efficiency of our hyperparameter tuning method for auto-tuning. Section 5.4.1 details the experimental setup. In Section 5.4.2, we discuss the efficacy of our method. In Section 5.4.3, we evaluate the performance of meta-strategies to optimize and expand the hyperparameter tuning. We then evaluate extensive non-exhaustive tuning in Section 5.4.4. Finally, we evaluate the feasibility and efficiency of our approach in Section 5.4.5.

5.4.1 Experimental Setup

To obtain a diverse set of real-world cases for evaluation, we use the four applications auto-tuned on six GPUs as described in Section 5.3.4, resulting in 24 unique search spaces. Each application has been fully brute-force auto-tuned on each GPU system as described in Section 5.3.4, while all evaluations are performed on the DAS-6 supercomputer [8]. The nodes in DAS-6 have a 24-core AMD EPYC-2 7402P CPU, 128 GB of RAM, and are running Rocky Linux 8 with Linux kernel version 4.18. The Python version used is 3.11.7, and Kernel Tuner at 1.3.0. 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

Table 5.3: Hyperparameter values for the optimization algorithms. Optimal values are in bold, those closest to mean in italics.

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*}
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}
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}

MI250X, Nvidia A100, and Nvidia A4000, and the test set consists of twelve search spaces resulting from the four applications on the AMD W6600, AMD W7800, and Nvidia A6000. These combinations of applications and devices result in a highly diverse set of search spaces as seen in [99].

While Kernel Tuner provides a plethora of optimization algorithms as seen in Table 5.1, due to space constraints, we select the four optimization algorithms shown in Table 5.3 to represent the diverse range of global optimization algorithms and hyperparameters available. *Simulated Annealing* explores the search space by probabilistically accepting worse solutions to escape local optima, gradually reducing this randomness over time. *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 the W hyperparameter of PSO had no meaningful effect on the score, and as such is left out. While several of the hyperparameters used in Table 5.3 are numerical and thus have a much larger set of possible values, we limit the values to those shown in order to conduct an exhaustive search of all possible combinations. This allows for determining the potential performance impact of hyperparameter tuning and enables a fair evaluation of meta-strategies on this exhaustive data. Each combination of hyperparameter values shown in Table 5.3 is run 25 times on each of the 12 search spaces to obtain a robust performance score. This means that tuning the hyperparameters of e.g. Genetic Algorithm as in

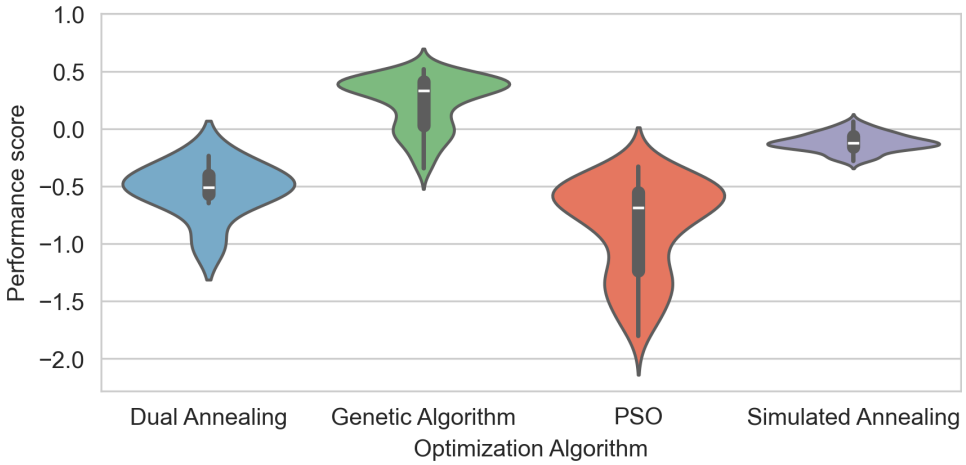


Figure 5.2: Violin plots of the performance scores (higher is better) for all hyperparameter configurations of the evaluated optimization algorithms, showing the mean (white line), boxplot (black box), and distribution (area).

Table 5.3 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, as discussed in Section 5.3.2. In the following performance comparisons, each instance is re-executed with 100 repeats to mitigate stochasticity.

5

5.4.2 Hyperparameter Tuning Efficacy

First, we discuss the results regarding the impact and generalization of our hyperparameter tuning method.

Figure 5.2 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 baseline, a negative indicates worse performance than the baseline, and a score of 1 indicates that the optimum is consistently found immediately - the ultimate goal of any optimization algorithm. For more information on the performance scores, see Section 5.3.2. The violin plots in Fig. 5.2 depict large differences in scores between the various hyperparameter configurations of each optimization algorithm. In addition, it shows the difference in sensitivity to hyperparameter tuning between optimization algorithms; e.g. PSO is substantially more sensitive than simulated annealing. An average difference in performance score of 0.865 between the best and worst hyperparameter configurations highlights the substantial performance improvement potentially gained.

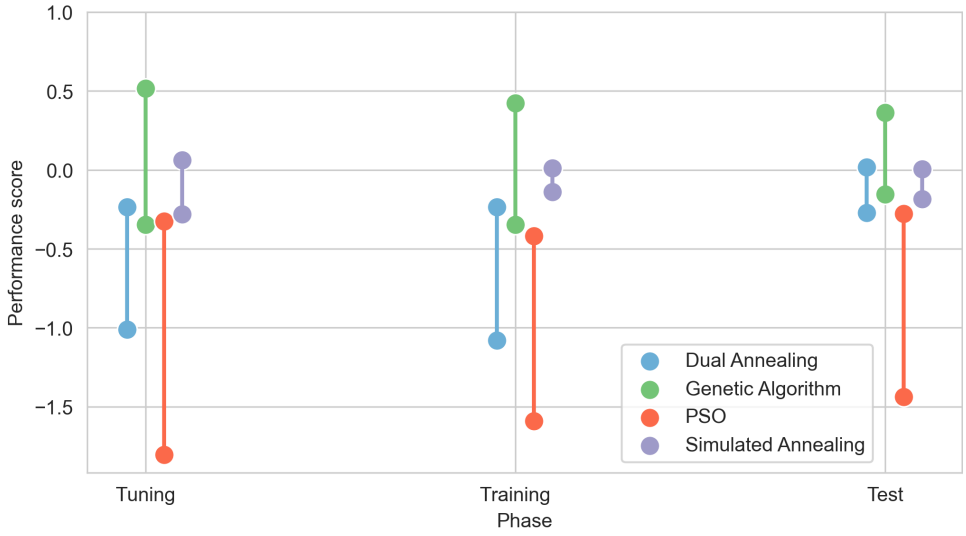
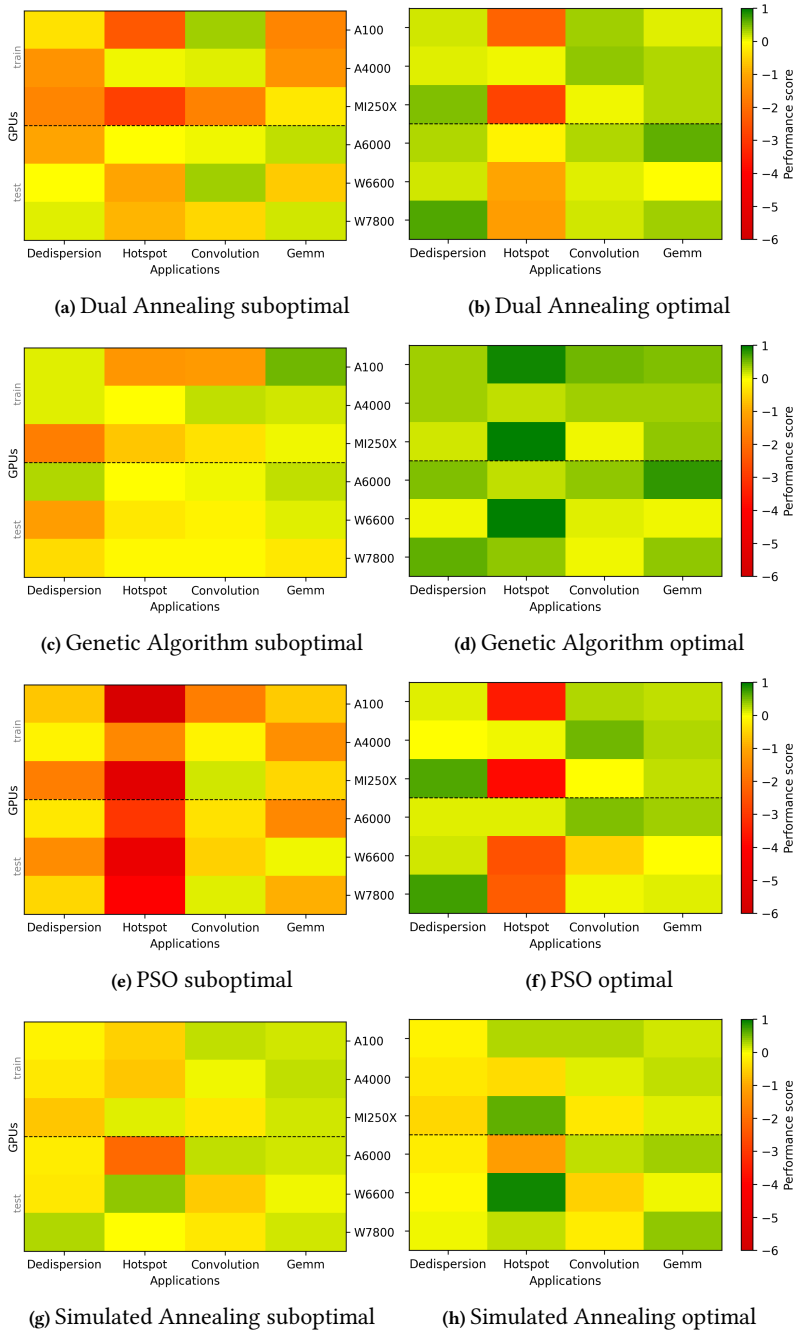


Figure 5.3: Best and worst scores on tuning, training (re-executed for comparison), and test for evaluated optimization algorithms.

We validate this by examining the overall performance and generalization of the hyperparameter tuning across the search spaces. Figure 5.3 shows that the performance scores of the best- and worst-performing hyperparameter configurations obtained in tuning, which were executed 25 times during tuning, remain largely stable when re-executed with 100 repeats on the same training data. Most importantly, the best configuration performance is also comparable on the test set of search spaces not tuned on, indicating excellent generalization performance.

To further verify the generalization performance, we can compare the performance on the 24 individual search spaces. In Fig. 5.4, visual inspection of both the suboptimal (worst performing, left column) and optimal (best performing, right column) versions of each optimization algorithm reveals that each of the optimal versions improves upon their counterpart in general, instead of over-optimizing on a limited number of search spaces to boost the score. Interestingly, all optimization algorithms, except for the optimal versions of *genetic algorithm* and *simulated annealing*, struggle with the hotspot application search spaces. Most importantly, it can be seen that for the vast majority of search spaces in both the training (upper half of each plot) and test sets (lower half of each plot), the performance is substantially improved.

Finally, we compare the optimal hyperparameter configuration to the most average-performing hyperparameter configuration (i.e., closest to the mean score in Fig. 5.2) to gauge the average impact of hyperparameter tuning. Figure 5.5 compares these optimal



(a) Dual Annealing suboptimal

(b) Dual Annealing optimal

(c) Genetic Algorithm suboptimal

(d) Genetic Algorithm optimal

(e) PSO suboptimal

(f) PSO optimal

(g) Simulated Annealing suboptimal

(h) Simulated Annealing optimal

Figure 5.4: Impact of tuning on optimization algorithm performance; left-hand and right-hand columns show the performance on all search spaces for suboptimal and optimal optimization algorithm versions, respectively.

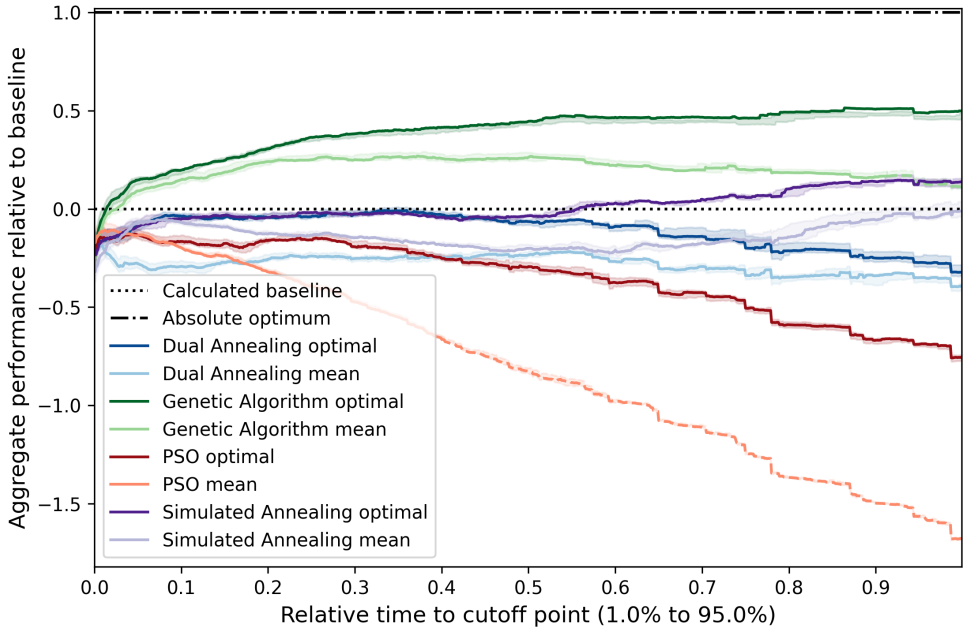


Figure 5.5: Aggregate performance over time between optimization algorithms with mean and optimal hyperparameters across all search spaces.

and average configurations of each optimization algorithm over time across all the search spaces, showing that the optimal optimization algorithms outperform their average counterparts by a wide margin. Quantifying this difference with the performance score, Dual Annealing is improved by 0.170, Genetic Algorithm by 0.192, PSO by 0.473, and Simulated Annealing by 0.149, for an average improvement of 94.8% over the average hyperparameter configuration.

5.4.3 Meta-strategies for Hyperparameter Tuning

With the efficacy of hyperparameter tuning established, we evaluate the usage of meta-strategies to efficiently explore a more expansive hyperparameter search space as opposed to exhaustive hyperparameter tuning.

By running various meta-strategies on the exhaustively evaluated hyperparameter tuning spaces for each of the algorithms and applying the performance scoring once more, we can investigate whether hyperparameter configurations in auto-tuning can be effectively optimized with optimization algorithms themselves. These meta-strategies can be the same optimization strategies that are already included with Kernel Tuner; for simplicity, we will reuse the optimization algorithms tuned earlier as meta-strategies. Figure 5.6 depicts the results, with all optimization algorithms performing very well after an initial

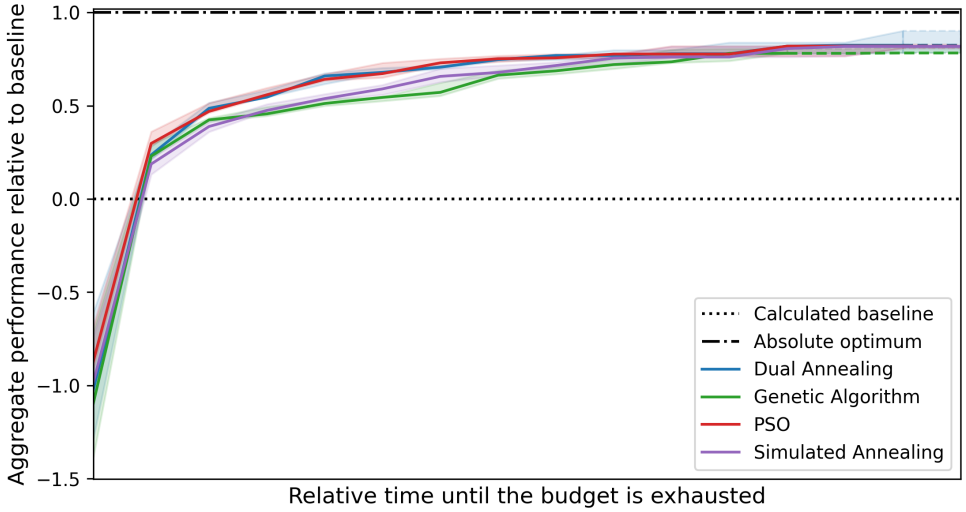


Figure 5.6: Aggregate performance over time for various meta-strategies on the hyperparameter tuning search spaces, 100 repeated runs each.

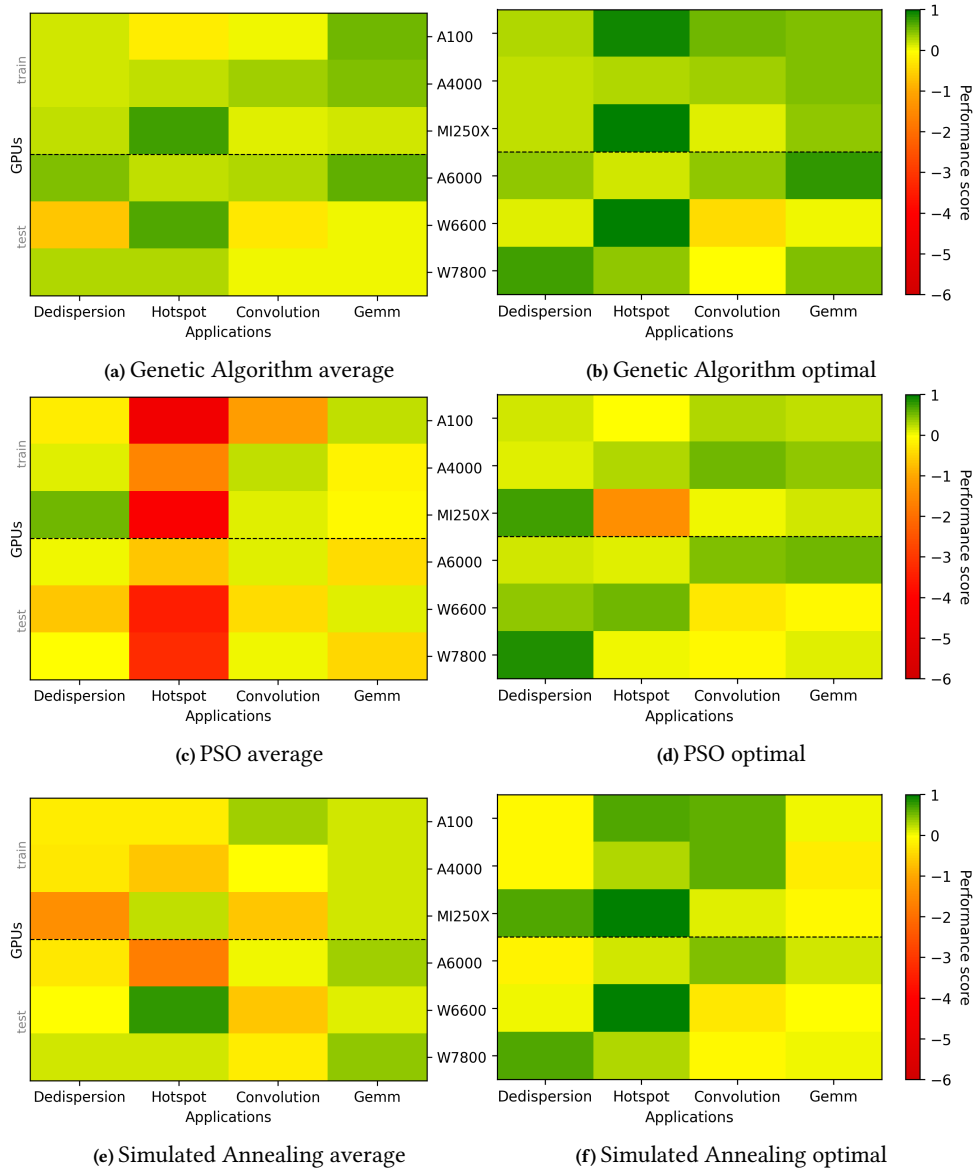
Table 5.4: Extended hyperparameter values for the optimization algorithms. Optimal values are in bold.

Algorithm	Hyperparameter	Values
Genetic Algorithm	method	{**single_point**, two_point, uniform, disruptive_uniform}
	popsize	{ 2, 4, 6, . . . , **26**, . . . , 50 }
	maxiter	{ 10, 20, 30, . . . , **150**, . . . , 200}
	mutation_chance	{ **5**, 10, 15, . . . , 100 }
Particle Swarm Optimization (*PSO*)	popsize	{ 2, 4, 6, . . . , **50** }
	maxiter	{ 10, 20, 30, . . . , **190**, 200 }
	c1	{ 1.0, 1.25, 1.5, . . . , **3.5** }
	c2	{ 0.5, 0.75, **1.0**, . . . , 2.0 }
Simulated Annealing	T	{ **0.1**, 0.2, 0.3, . . . , 2.0 }
	T_min	{ **0.0001**, 0.0011, . . . , 0.1 }
	alpha	{0.9925, 0.995, **0.9975**}
	maxiter	{ **1**, 2, 3, . . . , 10 }

startup cost, with an average performance score of 0.223. These findings demonstrate the substantial increase in efficiency of using a meta-strategy over random or exhaustive search, which allows much wider ranges of hyperparameter values to be tuned with a high probability that a near-optimal hyperparameter configuration is found.

5.4.4 Extended Hyperparameter Tuning

Having established that hyperparameter tuning auto-tuning optimization algorithms generalizes well and is effective, and meta-strategies are efficient at finding near-optimal hy-



(a) Genetic Algorithm average

(b) Genetic Algorithm optimal

(c) PSO average

(d) PSO optimal

(e) Simulated Annealing average

(f) Simulated Annealing optimal

Figure 5.7: Impact of extended tuning on optimization algorithm performance; left-hand and right-hand columns show the performance on all search spaces for average and optimal optimization algorithm versions, respectively.

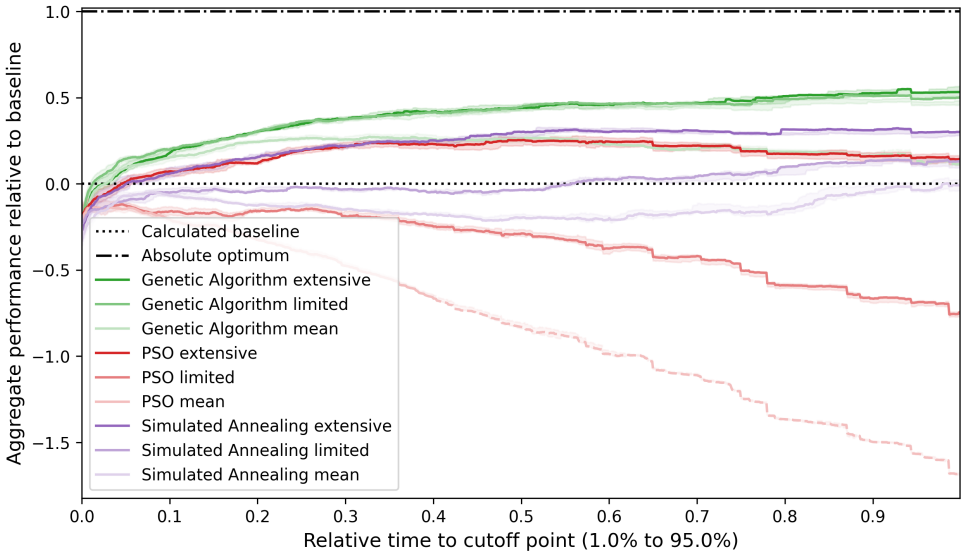


Figure 5.8: Aggregate performance over time between optimization algorithms with mean and optimal (limited and extended) hyperparameters across all search spaces.

perparameter configurations, we perform a final extensive, non-exhaustive hyperparameter tuning on the optimization algorithms with numerical hyperparameters exposed for a more realistic approach to hyperparameter tuning. This extensive hyperparameter tuning is ran on each optimization algorithm for 7 days using Dual Annealing as a meta-strategy. The hyperparameter values are shown in Table 5.4.

We can compare the optimal hyperparameter configuration of this extended tuning to the most average-performing hyperparameter configuration of Section 5.4.2 to gauge the average impact of a realistic hyperparameter tuning scenario. The improvements between the average configuration of the limited tuning and the optimal configuration of the extended tuning, seen for each search space in Fig. 5.7, indicate that the hyperparameter tuning enhances overall performance (on both train and test), like before in Fig. 5.4. Figure 5.8 compares the optimal and average configurations of each optimization algorithm over time across all the search spaces, showing that after the optimization algorithms with extended tuning outperform their exhaustive counterparts by an even wider margin than previously with the limited exhaustive hyperparameter tuning. Quantifying this difference with the performance score, Genetic Algorithm is improved by 0.195, PSO by 1.002, and Simulated Annealing by 0.366, for an overall average improvement of 204.7% over the average limited hyperparameter configuration, and 210.8% on the test set of search spaces.

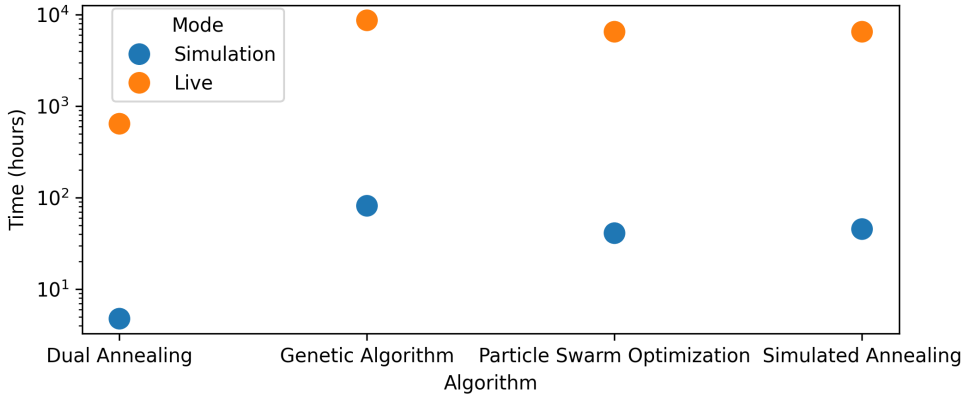


Figure 5.9: Tuning time comparison between live and simulation mode.

5.4.5 Feasibility with Simulation Mode

Finally, to evaluate the efficiency of our approach, we compare live tuning and using the simulation mode as shown in Fig. 5.9, where lower is better. We calculate the time live tuning would have taken by taking the 95% time budget of Section 5.4.1 for each search space in seconds, multiplied by the number of hyperparameter configurations for each algorithm and the number of repeats (25 as per Section 5.4.1). With our method, exhaustive hyperparameter tuning of the limited set of Table 5.3 took approximately 4.5 hours for the optimization algorithm with the least hyperparameter configurations (*Dual Annealing*) and 81 hours for the optimization algorithm with the most hyperparameter configurations (*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 exhaustive 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 22323 hours (nearly three years), whereas it took 172 sequential hours using the simulation mode, a 130x speedup.

5.5 Conclusion

In this work, we have introduced a novel approach to hyperparameter tuning for optimization algorithms used in auto-tuning, a hitherto often overlooked domain. As processor architectures and applications continue to grow in complexity, efficient auto-tuning is crucial to fully leverage the computational power of modern architectures, which in turn

depends on optimization algorithm performance. Our method addresses the challenges of hyperparameter optimization in this domain by building upon our statistically robust methodology for comparing optimization algorithms, introducing a simulation mode to make hyperparameter tuning feasible, and promoting the use of reproducible and shareable benchmark resources.

In this first evaluation of hyperparameter tuning for auto-tuning, the results show that our approach substantially improves the efficiency of the optimization algorithms in finding near-optimal configurations in auto-tuning search spaces and that these improvements hold on search spaces not trained on. By systematically tuning a limited set of hyperparameters for these algorithms with a robust performance score, we improved their overall performance by 94.8% on average, and extensive tuning extended this to an average improvement of 204.7%. 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. We show that the hyperparameters can be optimized efficiently using meta-strategies, allowing vast hyperparameter spaces to be tuned efficiently. The FAIR dataset of auto-tuning data provided in conjunction with this work lowers barriers to entry in auto-tuning research, as well as energy consumption and resource costs. Based on the evaluation, we conclude that our method for hyperparameter tuning of auto-tuning optimization algorithms has a substantial effect on the performance, even on a limited set of hyperparameters, and generalizes well beyond the training data. By addressing the efficacy, efficiency, and reproducibility challenges in hyperparameter tuning, this work contributes to the advancement of auto-tuning, enabling more efficient utilization of modern processors in scientific and industrial applications.

While our approach achieves substantial performance improvements across multiple real-world applications and architectures, the reliance on exhaustively explored search spaces to ensure statistical robustness may limit applicability in cases where such data is unattainable. Future work will explore methods for extending this approach to partially explored or dynamically generated search spaces.

The implementation and optimized hyperparameters in this work are included and set as defaults in Kernel Tuner.

6 Exploring the Application of Constrained Optimization

> *The more constraints one imposes,*
> *the more one frees one's self.*
>
> Igor Stravinsky

Many auto-tuning problems involve large discrete search spaces where many configurations are invalid due to hardware or correctness constraints. Standard evolutionary algorithms often waste effort evaluating such configurations. We extend four common evolutionary algorithms with constraint-awareness, enabling them to avoid infeasible solutions and converge more quickly. Experiments on representative benchmarks show faster convergence, higher-quality solutions, and consistent outperformance of state-of-the-art methods such as pyATF. The proposed algorithms are released as open-source contributions to Kernel Tuner, facilitating future research and adoption.

This chapter is based on "Constraint-aware Optimization in Auto-Tuning" by Willemsen, Heldens, van Nieuwpoort, and van Werkhoven [169].

6.1 Introduction

A key problem in auto-tuning is the fact that not all code variants constitute feasible (or valid) implementations. Modern massively parallel architectures are highly complex with deep memory hierarchies and heterogeneous compute cores, which introduce dependencies between different tunable parameters in the code. For example, when applying loop blocking, the tile size of the outer loop has to be a multiple of the tile size used in the inner loop. Auto-tuning frameworks therefore allow users to specify *constraints* along with the tunable parameters of their applications. Such constraints significantly complicate the exploration process and impose additional challenges on optimization algorithms [125, 68], in which a key problem is that many variants violating the constraints cannot be evaluated due to hardware limitations or software correctness requirements.

Existing optimization algorithms that do not explicitly handle constraints may waste significant computational resources exploring invalid or suboptimal configurations which can greatly reduce overall performance and efficiency. Effectively handling constraints within auto-tuning is therefore crucial, yet remains challenging due to the inherent blindness of classical optimization methods, such as evolutionary algorithms (*EAs*) and other metaheuristics, to the feasibility of generated solutions. Traditional optimization operators typically produce candidate solutions irrespective of constraint satisfaction, necessitating specialized constraint-handling techniques, including penalty methods, constraint-specific operators, or repair mechanisms [32]. While constrained optimization techniques have clear benefits, their impact on the performance of optimization algorithms for auto-tuning has, to the best of our knowledge, not yet been studied.

To this end, our work addresses the critical challenge of efficiently incorporating constraint-awareness into auto-tuning optimization algorithms and studying the impact of these techniques on optimization algorithm performance. Specifically, we investigate the integration of constraint-handling capabilities into four widely-used evolutionary optimization algorithms (Simulated Annealing, Particle Swarm Optimization, Firefly, and Genetic Algorithm), to systematically avoid or repair invalid solutions during the search, and thereby enhance their performance when solving constrained optimization problems encountered in auto-tuning in particular.

This chapter presents a real-world application study of constrained optimization for the auto-tuning domain. In particular, we make the following contributions:

- We review the uptake of techniques developed for constrained optimization in state-of-the-art auto-tuning frameworks.

- We present four evolutionary constrained optimization algorithms designed for automatic performance tuning as part of a generic auto-tuning framework.
- We quantify the performance impact of using constrained optimization algorithms over traditional optimization algorithms for a representative benchmark [159] set of auto-tuning problems, demonstrating substantial performance improvements across various tuning scenarios.
- Finally, we present a performance comparison of our methods with pyATF [135], a recently published state-of-the-art framework for constraint-based auto-tuning.
- We have implemented our methods as open-source contributions to Kernel Tuner, an easy-to-extend and widely-used open-source auto-tuning framework in Python.

The remainder of this chapter is organized as follows. Section 6.2 provides a high-level introduction to the auto-tuning problem and the need for constrained optimization in this domain. Section 6.3 reviews related work in constrained optimization as well as auto-tuning. Section 6.4 presents the implementation of our constraint-aware optimization algorithms for auto-tuning. Section 6.5 evaluates the impact of constraint-awareness on the performance of optimization algorithms. Section 6.6 concludes.

6.2 Background and Motivation

This section provides a general introduction to auto-tuning using an example kernel to illustrate how constraints arise when creating tunable applications for modern highly parallel architectures (such as, but not limited to, GPUs). In this chapter, we focus on compile-


```
__global__ void gemm(const float *A, const float *B, float *C,
                     int M, int N, int K,
                     float alpha, float beta) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < M && col < N) {
        float sum = 0.0f;
        for (int e = 0; e < K; e++) {
            sum += A[row * K + e] * B[e * N + col];
        }
        C[row * N + col] = alpha * sum + beta * C[row * N + col];
    }
}
```

Listing 4: Example GEMM kernel in HIP/CUDA.

time auto-tuning where the application can be tuned as part of the compilation process, as opposed to run-time auto-tuning where the application is tuned while it is running in production [120].

We will use general dense matrix-matrix multiplication (GEMM) as our example kernel. GEMM is part of the BLAS linear algebra specification, and is a fundamental and widely-used routine in high-performance computing and AI workloads. GEMM implements the multiplication of two matrices, A and B:

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

where α and β are scalars and C is the output matrix. A is of size $M \times K$, B is of size $K \times N$, and C is of size $M \times N$.

Listing 4 shows a simple implementation of GEMM as a GPU kernel in HIP/CUDA. This kernel can be executed on a massively parallel GPU processor by a large number of threads in parallel. Specifically, a two-dimensional array (x, y) of threads of size $M \times N$ allows each element in C to be computed by one thread.

GPU programming models, such as HIP or CUDA, do not allow the application developer to directly specify the total number of threads. Instead, threads are organized into *thread blocks*, and it is required to specify the block dimensions and the total number of blocks, also referred to as the *grid dimensions*. This means that the total number of threads may exceed the number of elements in C, which is why in Listing 4 a check is performed to prevent out-of-bounds array access.

For the GEMM kernel in Listing 4, the number of threads per block does not matter for the output of the kernel, but these numbers do impact the performance of the kernel. Thus, the thread block dimensions in both x and y are our first *tunable parameters* that constitute functionally equivalent variants of our implementation.

However, to ensure sufficient parallelism, each thread block must have a minimum size of, for example, 32 threads. In addition, most parallel architectures pose an upper bound on the number of threads per block, which can be queried before tuning. Typically, this limit is 1024 threads. These restrictions on the two parameters together form our first constraint:
`32 <= thread_block_x * thread_block_y <= 1024`. It is important to understand that this is a *hard constraint* as any candidate solution that exceeds 1024 threads per block cannot execute. Therefore, the execution time, i.e. the value of our cost function, cannot be obtained for candidate solutions violating this constraint.

The kernel shown in Listing 4 is a so-called naive kernel. A highly-optimized implementation would require extensive modifications, each introducing more tunable parame-

ters along with more constraints. For example, to efficiently exploit the cache hierarchy in modern architectures, we would need to introduce *loop blocking*. Loop blocking, in turn, introduces more constraints; for example, the loop count needs to be a divisor of the total work assigned to a thread block or a higher-level loop-blocking scheme.

Modern accelerators also have specialized memory spaces, such as *shared memory*, in which threads can stage data for later processing. Loop blocking is commonly applied to ensure efficient shared memory use. However, kernels that attempt to use more shared memory than provided by the hardware will fail to compile. Thus, constraints are used to exclude any solutions that would exceed the memory capacity.

Another commonly applied code transformation is *partial loop unrolling*, which is applied to improve instruction-level parallelism or to reduce register pressure in loops that would be fully unrolled by the compiler. However, a loop can only be partially unrolled to a degree that is a divisor of the total loop count, introducing another hard constraint.

Any highly-tunable kernel will have a relatively large number of tunable parameters, each with at least several values, and a large number of constraints that determine feasible candidate solutions. As a full review of all applied code transformation techniques and their constraints for a highly-tunable GEMM implementation is beyond the scope of this chapter, we refer the reader to Tørring et al. [159] for a more extensive description. For a comprehensive overview of code transformation techniques, we refer the reader to Hijma et al. [70].

6.3 Related Work

This section provides an overview of common approaches in constrained optimization and discusses their potential application to auto-tuning problems. In addition, we provide an overview of the application of constrained optimization in state-of-the-art auto-tuning frameworks.

6.3.1 Constraint-handling techniques

Constraint handling in metaheuristic optimization has been an active research area over the past decades. Broadly speaking, techniques fall into several categories, including penalty function methods, feasibility-based selection methods, repair mechanisms, and hybrid strategies combining these. We summarize representative work in each category, emphasizing methods applicable to discrete or combinatorial domains and their relevance to or applicability in auto-tuning. Unlike classic ordered combinatorial problems (e.g., TSP tours or graph structures) that often allow specialized operators to preserve feasibility, auto-tuning problems are typically defined using more general constraint functions.

One of the most common approaches is to incorporate constraint violations into the objective function via a penalty term. Early genetic algorithm (GA) research established static and dynamic penalty schemes that degrade an individual's fitness in proportion to its degree of constraint violation [32]. Static penalty methods use fixed penalties, while dynamic penalties increase the severity over generations. Adaptive penalty techniques go further by tuning penalty parameters based on feedback during the run. For example, Coello Coello [33] proposed a self-adaptive penalty approach where penalty factors evolve along with the population. This was shown to improve GA performance on engineering design constraints by reducing the need for manual penalty calibration. Tessema and Yen [157] developed adaptive penalty formulations that adjust penalties in response to the current population's feasibility ratio, aiming to maintain selection pressure without prematurely excluding infeasible regions. Penalty methods are general and straightforward to implement in any metaheuristic by modifying the cost function.

An alternative to numeric penalties is to modify the selection mechanism to prioritize feasibility. Deb [41] introduced a tournament selection method that gives preference to feasible solutions and, when comparing infeasible ones, favors those with smaller constraint violations. This approach does not require any penalty parameter and became a widely used baseline for GAs under constraints [92]. Runarsson and Yao [132] proposed the stochastic ranking method, which probabilistically intermixes objective and constraint violation rankings when sorting the population. Their approach effectively balances objective optimization and feasibility without a fixed penalty coefficient and was originally demonstrated on standard continuous benchmark problems. These ideas have influenced constraint handling in other metaheuristics as well, including differential evolution and ant colony optimization [178]. However, these approaches are difficult to apply in an autotuning context, because the objective function value cannot be obtained for solutions that violate the constraints. For example, a kernel that requires more shared memory than available on the GPU simply fails to compile, thus its execution time cannot be measured.

Especially in discrete and combinatorial domains, a common strategy is to repair infeasible solutions using a problem-specific heuristic. Instead of discarding or penalizing an infeasible offspring, the algorithm modifies it minimally to satisfy the constraints. Early GA systems for numerical constraints, such as GENOCOP [104], used specialized repair operators to project solutions back into the feasible region (e.g., solving linear constraint viola-

Table 6.1: Overview of support for constrained optimization in related and prior work. SA = Simulated Annealing. PSO = Particle Swarm Optimization.

Tool	Supports constraints	Search Space Representation	Optimization algorithms	Constraint handling
AUMA [47]	✗	list	K-Bagging Neural Networks	Only feasible solutions
CLTune [114]	✓	list	SA, PSO, Neural Network, Random, Exhaustive	Only feasible solutions
OpenTuner [3]	✗	list	SA, PSO, Bandit, various Evolutionary Algorithms, Local Search, Random, Exhaustive	Left to the user
ytopt [175]	✓	ConfigSpace	Bayesian Optimization	Left to the user
GPTune [95]	✓	scikit-optimize.space	Bayesian Optimization	Single constant penalty
KTT [120]	✓	chain-of-trees	Markov Chain Monte Carlo, Profile-based search, Random, Exhaustive	Only feasible solutions
ATF [125] pyATF [135]	✓	chain-of-trees	SA, Differential Evolution, Bandit, Local Search, Random, Exhaustive	Only feasible solutions
BaCO [68]	✓	chain-of-trees	Bayesian Optimization, Exhaustive	Only feasible solutions. Surrogate model for hidden constraints.
Kernel Tuner (2019) [165]	✓	list	20 different global and local optimization algorithms, including Annealing methods, Genetic/Evolutionary methods, Swarm-based methods, and Bayesian Optimization	Single constant penalty or only feasible solutions

tions) rather than relying on penalties. In combinatorial optimization, repair-based operators have been successfully employed for scheduling and routing problems. For example, Dorn and Girsch [45] describe a GA for steel production scheduling where crossover and

mutation are designed to detect and repair any constraint violations (using domain knowledge) immediately, thus always producing solely feasible schedules. Mitchell et al. [105] introduced the "GeneRepair" operator for the TSP, which is another illustrative case: after standard crossover, if a tour is invalid (e.g. duplicate cities), the operator swaps in missing cities to restore feasibility, which led to faster convergence to good tours. Repair mechanisms can exploit the structure of a problem's constraints (like the permutation structure in TSP or precedence constraints in scheduling), making them powerful for those domains. However, in a generic auto-tuning framework that allows users to define constraint functions that are specific to the tuning problem at hand, repair can be difficult and may require domain-specific knowledge regarding the violated constraint. The upside of repair is that the search operates mostly in the feasible space, but the downside is the need for custom repair logic per problem and the risk of biasing the search or reducing diversity if repairs always follow the same pattern.

Researchers have also combined strategies for constrained metaheuristic search. Multi-objective techniques treat the original objective and constraint violations as separate objectives, using evolutionary multi-objective optimization principles to push the population toward the Pareto front of minimizing both cost and infeasibility. Surry et al. [150] introduced this principle in a genetic algorithm, where solutions are first sorted based on the nondomination of their constraint violations, and then also ranked based on their objective function value. Coello Coello and Mezura-Montes [31] proposed a Pareto dominance-based tournament selection method for a genetic algorithm. Similarly, Venkatraman and Yen [162] used such multi-objective prioritization of feasibility with dynamic population sizing. He and Wang [64] applied this idea to Particle Swarm Optimization, using multiple swarms that co-evolve both the solution space and the penalty space. While these methods elegantly integrate the constraint satisfaction problem with the optimization of the objective function, they are difficult to apply to auto-tuning where the objective function values of infeasible solutions cannot be obtained.

Constraint handling in metaheuristics has evolved from basic penalty functions to sophisticated mechanisms like adaptive penalties, multi-objective formulations, and co-evolutionary repair strategies. For auto-tuning, these techniques provide a toolkit to navigate a search space that is often irregular and discontinuous. However, the application and effectiveness of these techniques to the constrained optimization problem of auto-tuning have hardly been explored so far.

6.3.2 State of the art in Constrained Optimization for Auto-Tuning

In Table 6.1 we provide an overview of the use of constrained optimization methods in auto-tuning frameworks. What most auto-tuning frameworks that support user-defined constraints have in common is that the feasibility of candidate solutions is checked as part of the search space construction method. ATF [125], KTT [120], BaCO [68], and pyATF [135] use a structure called chain-of-trees to store the set of valid configurations in memory.

AUMA [47] and OpenTuner [3] simply use a list to represent the search space. AUMA [47] does not support constraints directly but relies on an external tool to generate the list of valid configurations. OpenTuner [3] does not support constraints.

CLTune [114] maintains the full list of feasible configurations in memory but does support constraints. This list is generated when the tuning process starts by iterating through the entire Cartesian problem space and checking, for each possible solution in the search space, whether any of the constraints are violated. CLTune implements SA, PSO, and a Neural Network-based search algorithm in addition to exhaustive and random search. The search space representation is used to ensure only valid points are evaluated. For example, when using PSO, the movement of particles is limited to only valid neighboring configurations. The particle movement step is simply repeated until a valid configuration is found. There is also a probability that the particle does not move at all. To check the validity of configurations, CLTune simply iterates through the entire list of valid configurations until it finds the matching configuration, if the configuration is not found it is assumed to be invalid. Similarly, when moving to a neighboring configuration during Simulated Annealing, CLTune iterates through the entire list to find all neighbors of a configuration. Despite support for constrained optimization, none of the optimization algorithms implemented in CLTune outperformed random search in their evaluation [114].

GPTune and ytopt rely on `scikit-optimize.space` and `ConfigSpace`, respectively, which represent multidimensional configuration spaces, but do not enumerate or store individual configurations. Instead, these provide an interface to generate random samples from the search space, after which the validity of the drawn sample is checked. As constraint resolution is not supported by `scikit-optimize.space`, GPTune relies on an additional internal check on sampled points. OpenTuner [3] and ytopt are designed as libraries that allow users to implement auto-tuners. As such, the entire implementation of how to compile and benchmark possible solutions, as well as how to handle the evaluation of invalid configurations is left to the users of these frameworks. For example, some examples of ytopt show that infinity is returned instead of the execution time when a selected configuration does not compile successfully.

ATF [125] introduced the chain-of-trees search space representation and construction method that has since been adopted by many auto-tuning frameworks. The same authors have recently presented pyATF [135] as a new state-of-the-art framework for constraint-based auto-tuning. pyATF and ATF both support a modest number of optimization algorithms, all of which operate on a continuous domain $[0,1]^N$, where N is the number of tunable parameters. The objective function maps the continuous coordinates back to the nearest valid configuration in the chain-of-trees representation. However, the construction of the chain-of-trees representation is known to be expensive as described in Chapter 4.

KTT [120] is a C++ auto-tuning framework with a focus on runtime auto-tuning. KTT also uses chain-of-trees and its optimization algorithms operate directly on the indices into the chain-of-trees. The algorithms are designed to only sample from the feasible solutions when performing random sampling or retrieving the list of neighboring configurations.

BaCO, like ATF, samples only from valid configurations based on the user-defined constraints. However, BaCO takes an interesting approach to also learn so-called *hidden constraints*. Some configurations might appear to be valid according to user-defined constraints, but turn out to be infeasible during compilation or at run time, and are thus considered to violate hidden constraints. A separate surrogate model is trained to predict the feasibility of solutions according to the hidden constraints and the acquisition function is modified to take the probability of feasibility into account.

Kernel Tuner (2019) [165], prior to the extensions presented in this chapter, implements many different optimization algorithms, including Simulated Annealing, Particle Swarm Optimization, and Genetic Algorithm. To allow continuous optimization algorithms, such as Particle Swarm Optimization, to work on the discrete optimization problem of auto-tuning, Kernel Tuner maps the discrete points linearly into a continuous domain $[0,1]^N$, where N is the number of tunable parameters. First, the values of the tunable parameter with the largest set of possible values are linearly distributed across equidistant points in the domain $[0,1]$ with distance ε. In the other dimensions, values are mapped to points distributed between $[0, m \times \varepsilon]$, where m is the number of values in that dimension. This method ensures that regardless of the number of values in a particular dimension, a perturbation by ε in any dimension leads to a position that represents a different solution in the discrete space.

Kernel Tuner's objective function converts the continuous coordinates back to discrete points by snapping to the nearest discrete point. If the selected configuration is not valid, a simple static penalty is used when the solution violates one or more constraints [134].

Aside from this static penalty, the optimization algorithms in Kernel Tuner were not modified for constrained optimization.

6.4 Design and Implementation

This section presents an overview of the design and implementation of our methods implemented in Kernel Tuner, an open-source auto-tuning framework[1].

The overall auto-tuning objective is to determine the optimal code variant $v^\star$ (assuming maximization) as follows:

$$v^\star = \arg\max_{v \in \mathcal{V}} f_{H_j,I_k}(A_{i,v}) \tag{6.1}$$

Where we have an application A_i on a hardware platform H_j for an input data set I_k to maximize the performance measured by $f_{H_j,I_k}(A_i)$ over the code variants in $\mathcal{V}$.

6.4.1 Kernel Tuner

Kernel Tuner is generally used as an external framework for developers to benchmark and optimize GPU kernels in isolation, which can then be used with applications in any host programming language. An overview of the software architecture of Kernel Tuner

[1] https://github.com/KernelTuner/kernel_tuner

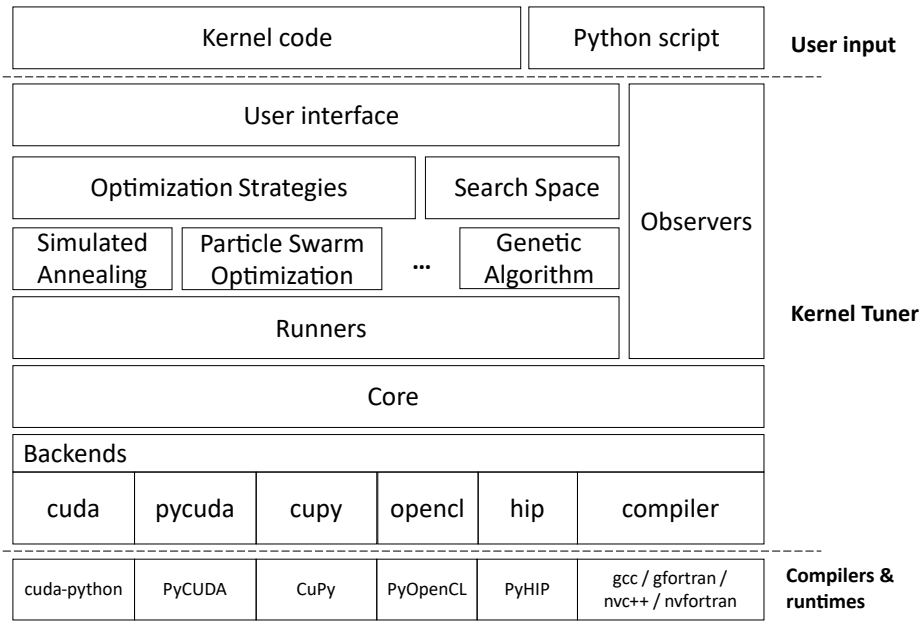


Figure 6.1: Overview of the software architecture of Kernel Tuner.

is shown in Fig. 6.1. Users of Kernel Tuner create a small Python script that points to the kernel code and describes the data set used for benchmarking, the tunable parameters that constitute the code variants, and the constraints. There are also various optional settings that users can specify, such as derived metrics to be computed, the optimization objective to use, which optimization algorithm to use, and hyperparameters to this optimization algorithm.

The tuner starts with the search space construction, which is further detailed below. The optimization strategy uses the search space to select new candidate solutions for evaluation. Some optimization algorithms have hyperparameters such as the annealing schedule, or the number of generations, which control when to stop the optimization process. However, Kernel Tuner interrupts an optimization algorithm when the user-specified optimization budget is exceeded. The runners are responsible for the actual evaluation of candidate solutions, which means compiling and measuring the performance of the code variants on the GPU. The performance of each candidate solution is measured multiple times and the average execution time is returned to the optimization algorithm.

The runner uses a single high-level interface inside Kernel Tuner's core layer that interfaces to the low-level backends. The backends, in turn, interface with the Python bindings of the compilers and device runtimes, such as CUDA, OpenCL, and HIP. Also shown in Fig. 6.1 are the Observers which facilitate the observation of metrics besides execution time, such as the energy consumed by the GPU or the numerical accuracy of the computation.

In Kernel Tuner, the auto-tuning search space construction problem is formalized as a Constraint Satisfaction Problem (*CSP*) defined by $\mathscr{P} = (X, D, C)$, where:

- $X = \{x_1, x_2, \ldots, x_n\}$ is a finite set of variables, each corresponding to a tuning parameter (e.g., block size, tile width).
- $D = \{D_1, D_2, \ldots, D_n\}$ is a set of finite domains, where D_i is the set of legal values for variable x_i.
- $C = \{c_1, c_2, \ldots, c_m\}$ is a finite set of constraints, where each c_j is a predicate over a subset of variables $\text{scope}(c_j) \subseteq X$ that restricts the allowable combinations of values based on hardware limits or algorithmic correctness.

A solution to the auto-tuning search space is then a total assignment $\mathscr{V} : X \to \bigcup D_i$ such that $\mathscr{V}(x_i) \in D_i$ for all i, and all constraints $c_j \in C$ are satisfied under $\mathscr{V}$. What distinguishes the auto-tuning problem from other constrained optimization problems is that the constraints in C are *hard constraints*, meaning that the performance function $f_{H_j,I_k}(A_{i,v})$ cannot be evaluated for any v that does not satisfy the constraints.

Kernel Tuner determines all configurations in the search space $\mathscr{V}$ before starting the tuning process. This is helpful as important search space characteristics, such as the true parameter bounds, can guide optimization algorithms more effectively and facilitate the use of stratified sampling techniques, such as Latin Hypercube Sampling as in Chapter 2. In addition, the full search space resolution substantially reduces the cost of valid neighbor lookups, which is important for several optimization algorithms that use these operations extensively. Thanks to the use of our optimized constraint satisfaction problem solver of Chapter 4, this step can be performed with minimal impact on the total execution time.

The *Search Space* object in Kernel Tuner provides various representations and operations on the constructed search space, using multiple internal representations for varying purposes, such as hash- and index-based representations for efficient lookups. Externally it provides a single interface for all search space-related operations that can be used by optimization algorithms to navigate the search space in a constraint-aware manner. To this end, *Search Space* contains functionality to query whether a particular solution is valid, to generate a number of random samples of only valid solutions, and to get all valid neighboring solutions of a particular solution.

There are different ways to define neighbors. Currently, we have implemented three definitions that specify when two solutions qualify as *neighbors*:

- **Strictly adjacent:** For every parameter, their values are identical or the previous/next value.
- **Adjacent:** For each parameter, their values are either identical or the nearest previous/next value that gives a valid configuration.
- **Hamming:** All parameters are identical except one.

The following subsections describe the implementations of the evolutionary optimization algorithms in Kernel Tuner. Specifically, we focus on how the algorithms use the search space to optimize the auto-tuning problem in the presence of hard constraints.

6.4.2 Simulated Annealing

The Simulated Annealing strategy in Kernel Tuner is a relatively straightforward implementation of Simulated Annealing with an annealing schedule based on a start temperature T, a minimum temperature T_{min} and α that controls the cooling rate from T to T_{min}. The annealing schedule that results from these parameters is scaled when the user also explicitly passes a maximum number of function evaluations, to ensure the algorithm uses the full budget. The algorithm starts from a randomly generated position.

Algorithm 2: Neighbor Generation for Simulated Annealing

Input: Current position s
$size \leftarrow |X|$; // Number of dimensions
$s_{new} \leftarrow [\,]$; // Initialize empty list
for $i \leftarrow 1$ **to** $size$ **do**
  if $r_1 < 0.2$; // Random $r_1 \in [0,1]$
  then
    $s_{new}[i] \leftarrow random\_choice(D_i)$;
  else
    $j \leftarrow \text{index}(s_i, D_i)$;
    if $r_2 > 0.5$; // Random $r_2 \in [0,1]$
    then
      $j \leftarrow j+1$;
    else
      $j \leftarrow j-1$;
    $j \leftarrow \min(\max(j,0), |D_i|-1)$;
    $s_{new}[i] \leftarrow D_i[j]$;
return s_{new};

Algorithm 3: Constraint-aware Neighbor Generation for Simulated Annealing

Input: Current position s, Search Space $\mathscr{V}$
$size \leftarrow |X|$; // Number of dimensions
$s_{new} \leftarrow s$; // Initialize from current position
for $i \leftarrow 1$ **to** $size$ **do**
  if $r < 0.2$; // Random $r \in [0,1]$
  then
    $s_{new} \leftarrow random\_neighbor(s_{new}, \text{'Hamming'}) \in \mathscr{V}$;
$s_{new} \leftarrow random\_neighbor(s_{new}, \text{'Adjacent'}) \in \mathscr{V}$;
return s_{new};

At each annealing step, the current position is perturbed to the next candidate solution with the procedure outlined in Algorithm 2. For each tunable parameter, there is a 0.2 probability that the current value in that dimension is changed to a random value of the tunable parameter, otherwise, the value is changed to a value directly adjacent to the current value.

The candidate solution is evaluated by compiling and benchmarking the corresponding code variant on the GPU. However, Kernel Tuner uses a memoization scheme to avoid evaluations of solutions that have already been compiled and benchmarked. This is relevant for the simulated annealing implementation because the annealing schedule is only advanced if the generated candidate actually gives new information. In other words, revisiting previously evaluated solutions does not count towards the number of function evaluations and is thus essentially 'free'. Because the optimization algorithm may get

Algorithm 4: Constraint-aware Repair Method for PSO

Input: Current position $x \in [0,1]^N$, Search Space $\mathscr{V}$
Procedure *Neighbors(p)*
  for $m \in$ {*'strictly-adjacent'*, *'adjacent'*, *'Hamming'*} **do**
    $n \leftarrow list\_neighbors(p,m) \in \mathscr{V}$;
    if $|n| > 0$ **then**
      return n;
  return [];
$p \leftarrow get\_params(x)$; // Convert from continuous space
$n \leftarrow$ NEIGHBORS$(p) \in \mathscr{V}$;
if $|n| > 0$ **then**
  $n \leftarrow to\_coordinates(n)$; // Convert to continuous space
  $s \leftarrow Euclidian\_distance(x,n)$;
  $n \leftarrow sort(n, key = s)$; // Sort neighbors on distance to x
  return $n[0]$;
return [];

stuck in an area where no new candidates are to be found, the algorithm restarts from a random position after 100 attempts.

To make the Simulated Annealing implementation in Kernel Tuner aware of search space constraints, we use the search space to only generate feasible solutions when generating a random sample at the start or restart of the algorithm. Furthermore, we adapt the perturbation of the current position to ensure that the returned candidate solution s_{new} is valid, using the procedure outlined in Algorithm 3. The algorithm queries the search space for neighbors of the current position using first Hamming and later adjacent neighbor relations as described in Section 6.4.1. A subtle difference with Algorithm 2 is that Algorithm 2 iterates through all dimensions and possibly changes to a Hamming neighbor in that dimension, while Algorithm 3 simply changes to a random Hamming neighbor at most $|X|$ times. Algorithm 3 is designed to mimic Algorithm 2. However, perfect replication is difficult due to the original implementation being unaware of search space constraints.

6.4.3 Particle Swarm Optimization (PSO)

The PSO strategy in Kernel Tuner is a simple and representative implementation of the PSO methods. Starting from a randomly generated sample population of solutions, the inertia, cognitive, and social coefficients are used to move the particles around according to their own best solution and the global best solution found so far. In contrast to Simulated Annealing, the particles in PSO are represented and move around in a continuous space.

When a solution needs to be evaluated, the continuous coordinates are mapped back to the discrete space of code variants using the method outlined in Section 6.3. The position of the particle is not adjusted as part of this process. However, when the nearest discrete point violates the constraints, a single constant penalty value is returned instead.

To make PSO efficiently deal with constraints on the auto-tuning search space we (i) use the search space to generate a population of only valid candidate solutions, which are then mapped back into continuous coordinates, and (ii) ensure that when the inverse conversion is needed during evaluation, the continuous coordinates are mapped back to only valid discrete configurations, using the procedure outlined in Algorithm 4. When the nearest discrete point is not feasible, we use the search space to retrieve a list of valid neighbors. We first try the *strictly-adjacent*, then *adjacent*, and then *Hamming* neighbor rules. All configurations in the first nonempty set that is returned are converted to continuous space and the closest point, by Euclidean distance in continuous space, is selected and used in the evaluation of the particle's position. Note that this repair method only affects evaluation and does not change the particle's position itself.

We also implement the Firefly Algorithm as a variant of PSO, using the same sampling and repair techniques to ensure only valid candidate solutions are evaluated.

6.4.4 Genetic Algorithm

The Genetic Algorithm strategy in Kernel Tuner uses a population of candidate solutions that is completely refreshed every generation. The Genetic Algorithm supports single-point, two-point, uniform, and disruptive uniform crossover. Individuals in the population are sorted based on the cost function value. Selection for crossover uses a beta probability distribution to ensure that individuals with better cost function values have a higher probability of being selected, as shown in Fig. 6.2. The advantage of this approach is that selection is biased towards better individuals independently of the magnitude of the objective function values.

Newly generated individuals also have a probability of being mutated. Mutation happens in exactly one parameter value, where the value is replaced with a different value for that tunable parameter, if any.

To improve the performance of the Genetic Algorithm in the presence of constraints, we use the search space to generate the initial population. After crossover, newly generated individuals may violate the constraints and therefore need to be repaired. Algorithm 5

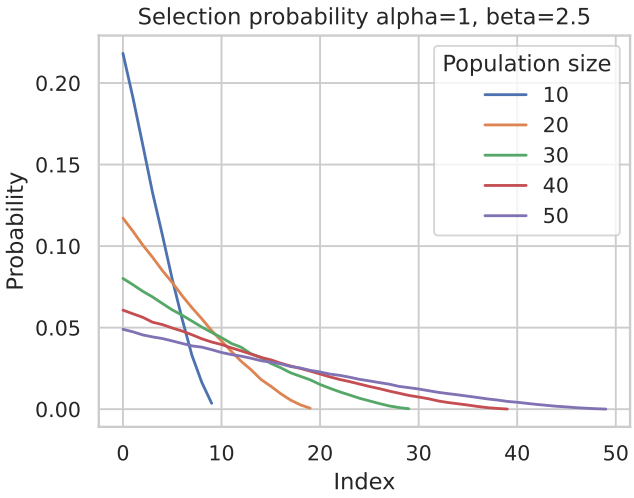


Figure 6.2: Beta-distributed relation between the index in the sorted population and the probability of being selected for crossover for different population sizes.

Algorithm 5: Constraint-aware Repair Method for GA

```
Input: Current solution s, Search Space 𝒱
if s ∉ 𝒱 then
    n ← Neighbors(s) ∈ 𝒱;
    if |n| > 0 then
        return random_choice(n);
return s;
```

shows the repair procedure used in GA. Reusing the Neighbors procedure from PSO (Algorithm 4), we try the *strictly-adjacent*, then adjacent, and then Hamming neighbor rules to retrieve the list of neighbors. The repair method replaces the invalid solution with a random neighbor in the first nonempty set that is returned. In contrast to PSO, the repaired configuration replaces the invalid individual in the population. Our constrained GA first repairs invalid solutions and then mutates if needed. Mutation is implemented to replace an individual with a random feasible Hamming neighbor to keep behavior similar to the original implementation while taking search space constraints into account.

6.5 Evaluation

In this section, we evaluate the effectiveness of the constrained optimization algorithms presented in Section 6.4 and quantify their performance improvement over classical opti-

Table 6.2: GPUs used in our experiments. *Only one out of two dies of the MI250X is used.

GPU	Year	Architecture	Cores	Memory	Cache	Bandwidth (GB/s)	Peak SP (GFLOP-S/s)
AMD W6600	2021	RDNA 2	1792	16 GB GDDR6	32 MB L3	224	10404
AMD MI250X*	2021	CDNA 2	7040	64 GB HMB2e	8 MB L2	1638	28160
AMD W7800	2023	RDNA 3	4480	32 GB GDDR6	64 MB L3	576	45250
Nvidia A4000	2021	Ampere	6144	8 GB GDDR6	4 MB L2	448	17800
Nvidia A6000	2020	Ampere	10752	48 GB GDDR6	6 MB L2	768	38710
Nvidia A100	2020	Ampere	6912	40 GB HMB2	40 MB L2	1555	19500

Table 6.3: Overview of the basic characteristics of the real-world applications.

Name	Cartesian size	Constr. size	Dimensions	No. constraints	No. values per parameter	Density (%)
Dedispersion	22272	11130	8	3	1 - 29	49.973
2D Convolution	10240	4362	10	4	1 - 16	42.598
Hotspot	22200000	349853	11	5	1 - 37	1.576
GEMM	663552	116928	17	8	1 - 4	17.622

mization algorithms. In addition, we present a comparison with pyATF, a state-of-the-art framework for constraint-based auto-tuning [135].

6

6.5.1 Experimental setup

For the evaluation, we focus on six different GPU models available in the DAS-6 [8] and LUMI [81] supercomputers. The GPU specifications are listed in Table 6.2. On DAS-6 we use Rocky Linux 8 with Linux kernel version 4.18, ROCM 6.0.2 with AMD clang 17.0.0, and CUDA 12.2 with GCC 9.4.0. For the MI250X, LUMI is running SUSE Linux 5.14.21, ROCM 5.2.3 with AMD clang 14.0.0. Note that the MI250X is a multi-chip module with two individually operating GPU dies, of which we use only a single die. The Python version used is 3.11.7. All measurements have been performed with our modified version of Kernel Tuner, which is available online. The pyATF version used is 0.0.9, the latest at the time of writing.

We will evaluate our approach using the BAT benchmark suite [159] of auto-tunable GPU kernels. Specifically, we use the *dedispersion*, *convolution*, *hotspot*, and *GEMM* kernels. These benchmark kernels are examples of widely used real-world applications in astronomy, image processing, materials science, and linear algebra respectively. The characteristics of these applications are shown in Table 6.3. The Cartesian size is the size of the complete combinatorial space without applications of constraints. The constrained size is the number of feasible solutions that remain after applying constraints. The number of dimensions equals the number of tunable parameters in the source code. Finally, the density is the percentage of valid solutions (constrained size over the Cartesian size). We have selected these search spaces to represent a wide range of densities and other characteristics, as we would like to study their influence on optimization algorithm performance for constrained optimization problems.

Dedispersion is a signal-processing kernel that reconstructs radio signals distorted by interstellar dispersion by applying a range of dispersion measures to time-domain samples across multiple frequency channels [137].

The *2D convolution* kernel performs image filtering by computing weighted sums over image regions, with tunable parameters for thread block size, work per thread, shared memory padding, and use of the read-only cache.

Hotspot is a thermal simulation kernel that estimates processor temperature by iteratively solving differential equations based on simulated power and initial temperature inputs, producing a temperature grid as output. This tunable implementation supports flexible thread/block configurations and temporal tiling.

GEMM (General Matrix-Matrix Multiplication) is the example operation that has been introduced in Section 6.2. We use the tunable GEMM kernel implemented in CLBlast [115], a tunable linear algebra library.

These applications also bring diversity in their performance characteristics; e.g. dedispersion and hotspot are generally bandwidth-bound, while convolution and GEMM are generally compute-bound. To obtain a diverse set of real-world auto-tuning cases for evaluation, we use these four auto-tuning applications on the six GPUs described in Table 6.2, resulting in 24 unique search spaces.

To compare the performance optimization algorithms, we use the methodology for comparing optimization algorithm performance for auto-tuning problems as outlined by the auto-tuning research community discussed in Chapter 3. This methodology provides a systematic approach to comparing optimization algorithms across auto-tuning search spaces. Per search space in the comparison set, the approach defines how to set the optimization budget and defines a *calculated performance baseline*, a statistical approximation

of *random search* over the feasible solutions. In particular, it defines a *performance score* $\mathscr{P}$ that quantifies an optimization algorithm's performance over the passed time relative to the calculated baseline, to have consistent, objective-independent, transparent, and comparable behavior across search spaces.

$$\mathscr{P}(\mathscr{F},A,H,I) = \frac{1}{|\mathscr{T}|}\sum_{t\in\mathscr{T}} \frac{\sum_{A_i\in A}\sum_{H_j\in H}\sum_{I_k\in I}\mathscr{P}(\mathscr{F},A_i,H_j,I_k)_t}{|A||H||I|} \tag{6.2}$$

The applications A, target hardware platforms H, and inputs I are the collections of A_i, H_j, and I_k of Eq. (6.1), $\mathscr{T}$ is the set of sampling points in time used to aggregate performance over time. This aggregate score enables robust comparison of optimization algorithms by capturing both the quality of the configurations found as well as the time taken to do so. As such, a difference when comparing two performance scores can indicate a difference in the quality of configurations found, the time taken to do so, or a combination of both. A score of 0.0 indicates that the performance over time is similar to the baseline, whereas a score of 1.0 indicates that the optimum is found immediately. For this evaluation section, 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. Each optimization algorithm has been run 100 times on each search space to mitigate stochasticity.

6.5.2 Impact of Constrained Optimization for Auto-Tuning

In this subsection, we investigate the impact of using constrained optimization techniques for auto-tuning applications. To this end, we compare the performance of Simulated Annealing (SA), Particle Swarm Optimization (PSO), Firefly Algorithm, and Genetic Algorithm (GA) with and without the modifications for constrained optimization presented in Section 6.4.

The hyperparameters of the optimization algorithms are shown in Table 6.4. Both the constrained and nonconstrained versions of the optimization algorithms use the same hyperparameters. The *maxiter* parameter in SA controls the number of local search steps within each annealing step. A limited hyperparameter tuning of the Genetic Algorithm revealed that single-point crossover works best for auto-tuning problems. The mutation chance is interpreted as a 'one in X' chance, so a mutation chance of 5 means there is a 0.2 probability of a mutation when generating offspring in the genetic algorithm.

Before comparing these optimization algorithms across all 24 search spaces, let us first compare the performance of the constrained-optimized versions of each optimization algorithm to their non-optimized counterpart on a single search space. Figure 6.3 compares

Table 6.4: Hyperparameters values for the optimization algorithms.

Algorithm	Hyperparameter	Values
Simulated Annealing (*SA*)	T	0.5
	T_min	0.001
	alpha	0.9975
	maxiter	2
Particle Swarm Optimization (*PSO*)	popsize	30
	maxiter	100
	w	0.5
	c1	3.0
	c2	0.5
Firefly Algorithm	popsize	20
	maxiter	100
	B0	1.0
	gamma	1.0
	alpha	0.2
Genetic Algorithm (*GA*)	method	single_point
	popsize	20
	maxiter	150
	mutation_chance	5

this over time in two plots for the GEMM kernel on the A4000. The top plot shows the absolute best-found lowest runtime on this search space for each of the algorithms, up to the absolute optimum of 13.09 milliseconds. The bottom plot shows the same data, but relative to the baseline (fixed to 0.0) and the optimum at 1.0, making it easier to distinguish performance differences among the optimization algorithms. As mentioned in Section 6.5.1, like all other experiments in this evaluation, the budget is set to the time it takes the calculated random search baseline to reach a configuration that has a performance of at least 95% of the distance between the search space median and optimum. In the bottom plot of Fig. 6.3, we can see that most algorithms start with negative scores, meaning that at the start the algorithms perform worse than the calculated baseline. The two Firefly algorithms exhibit early stopping behavior, denoted by the dotted line. It can be seen in Fig. 6.3 that each constraint-aware optimization algorithm outperforms its non-optimized counterpart by a significant margin. The performance difference for Simulated Annealing between its constrained and non-constrained variants is the smallest. However, for PSO, Firefly, and GA, the performance of the non-constrained variants approaches that of the random search baseline, while the constrained PSO, Firefly and GA variants perform substantially better.

To evaluate the overall performance, Fig. 6.4 compares the performance of the constrained-optimized versions of each optimization algorithm to their non-optimized counterpart over time across all 24 search spaces. While Simulated Annealing (*SA*) performs very

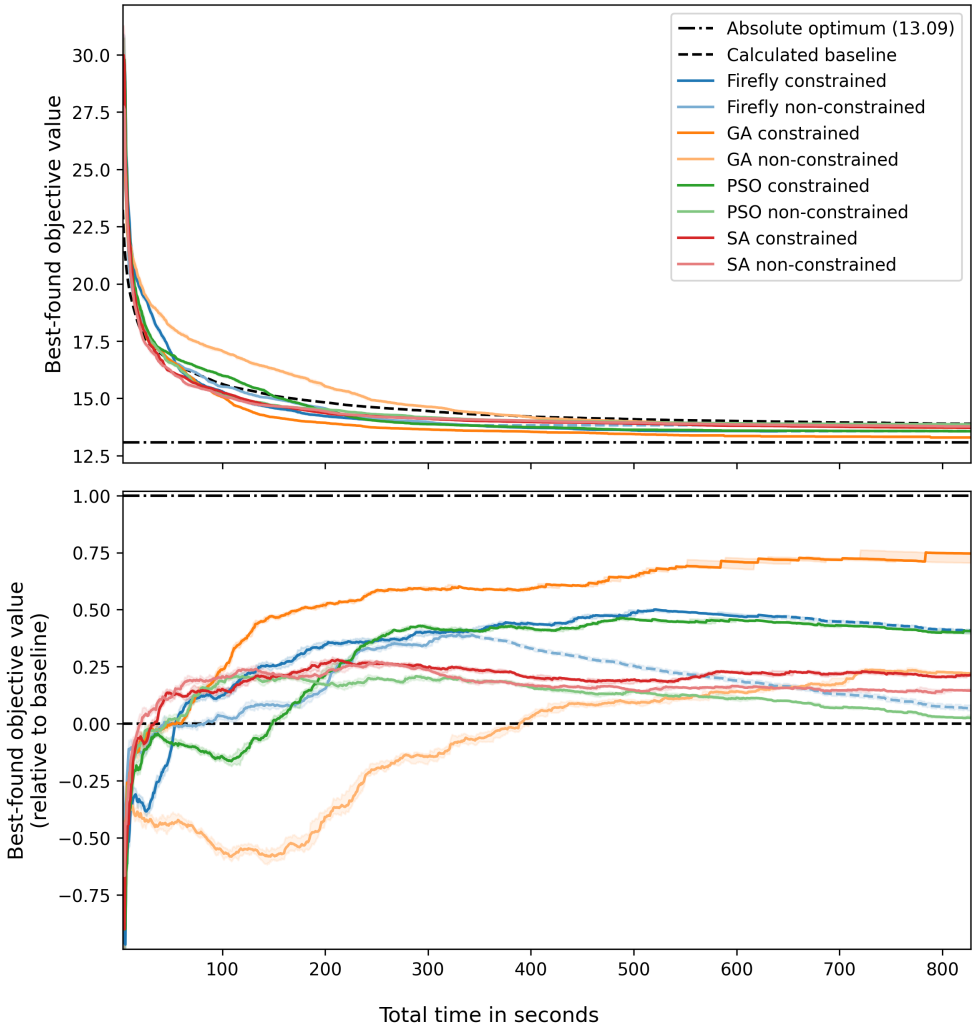


Figure 6.3: The performance over time of our constraint-aware optimization algorithms implementations and the Kernel Tuner default implementations for the GEMM kernel on the A4000.

similarly regardless of constraint awareness, the Genetic Algorithm (*GA*), Firefly, and PSO constraint-aware algorithms outperform their counterparts by a wide margin. Quantifying this difference with the *performance score* (an optimization algorithm's performance over the passed time relative to the calculated baseline), making Genetic Algorithm constraint-aware improved the score by 0.801, SA by 0.028, PSO by 0.964, and Firefly by 0.241.

Finally, we can compare the performance score per search space between our constraint-aware and the original versions for specific optimization algorithms to validate our find-

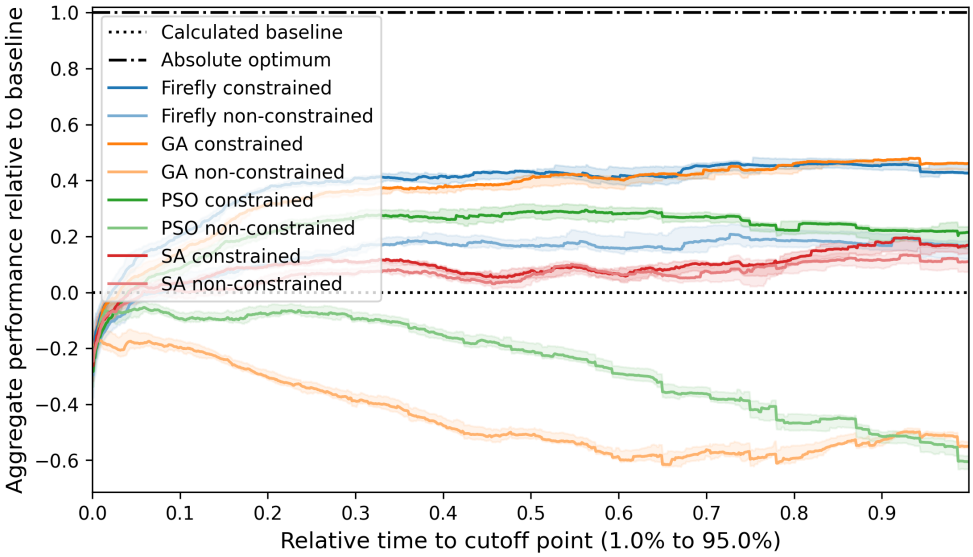
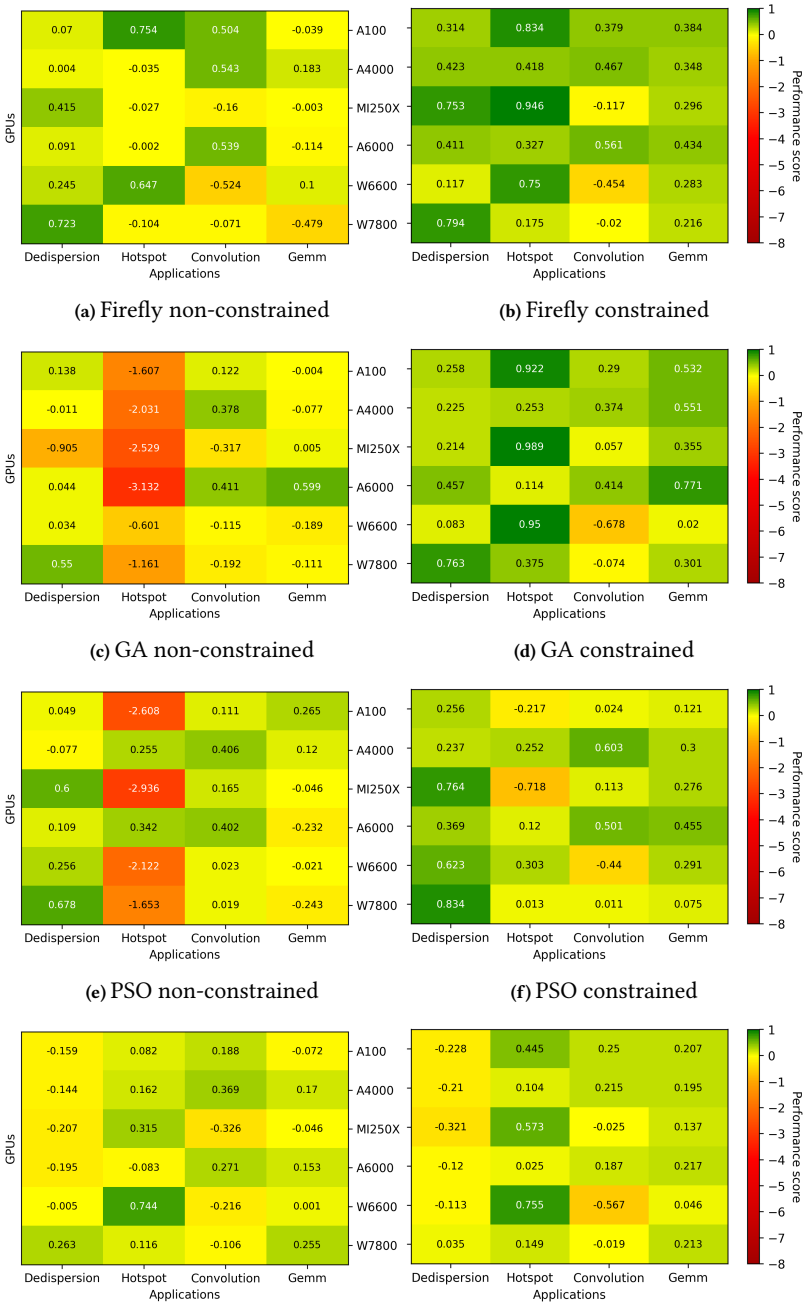


Figure 6.4: The aggregate performance over time of our constraint-aware optimization algorithms implementations and the Kernel Tuner default implementations.

ings, shown in Fig. 6.5. As would be expected when the performance improvement is due to constraint-awareness, the performance difference is greatest between the sparsest search spaces, especially *hotspot* and to a lesser extent *GEMM*, as per Table 6.3. This is particularly noticeable for GA and PSO, where the performance difference between both versions was also the largest in Fig. 6.4. While the performance difference between the constraint-aware and non-constrained version is less pronounced for SA in Fig. 6.4, comparing Fig. 6.5g to Fig. 6.5h shows that the constrained version performs substantially better on the sparser search spaces. Further investigation, using a performance profiler, revealed that the lack of overall performance difference is due to the increased computational cost of finding neighbors in contrast to the relatively straightforward non-constrained version.

6.5.3 Comparison with State of the Art

In this subsection, we evaluate the performance of our constraint-aware optimization algorithms against pyATF [135], a state-of-the-art framework for constraint-based auto-tuning. As both our methods and pyATF are implemented in Python, we can do a pure substitution of solely the optimization algorithms for a fair comparison. As it is known that the chain-of-trees search space generation can be expensive (see Chapter 4), we have implemented a retrieval mechanism to prevent this from influencing the results. In addition, pyATF does not have a memoization mechanism for retrieving previously evaluated configurations as



(a) Firefly non-constrained **(b)** Firefly constrained

(c) GA non-constrained **(d)** GA constrained

(e) PSO non-constrained **(f)** PSO constrained

(g) SA non-constrained **(h)** SA constrained

Figure 6.5: Impact of constraint-awareness on optimization algorithm performance per search space.

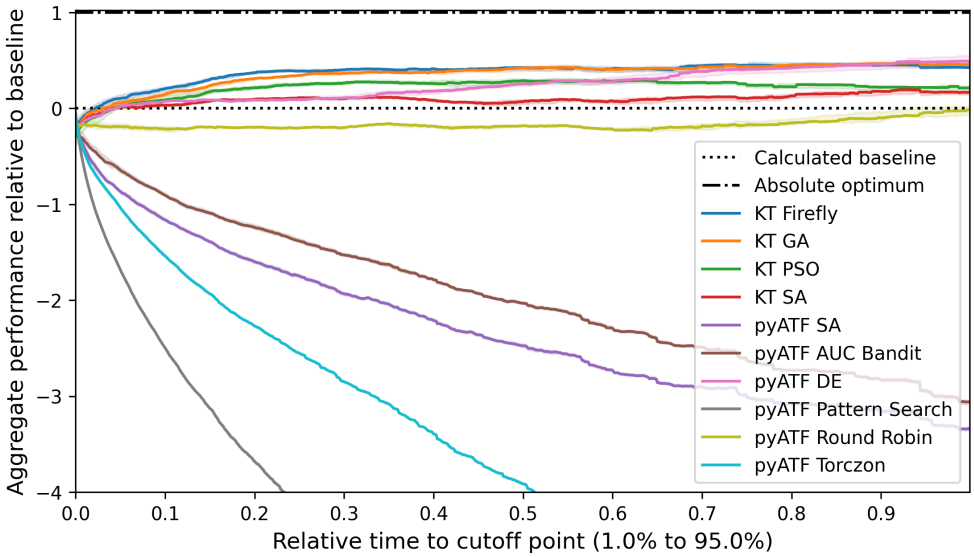
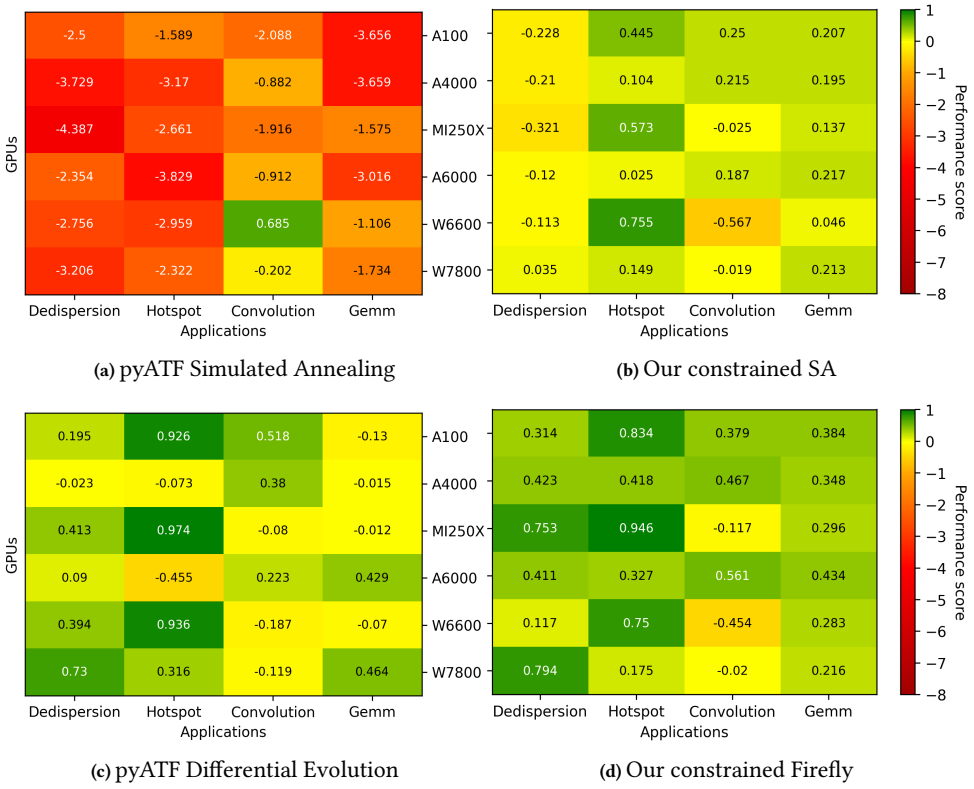


Figure 6.6: The aggregate performance of various pyATF and our constraint-aware optimization algorithms across all 24 search spaces. The plot has been cut off at -4 to improve legibility, *pyATF Torczon* and *pyATF AUC Bandit* continue to -6 and -10 respectively.

Kernel Tuner does. To ensure a fair comparison of only the optimization algorithms in both frameworks, we have used Kernel Tuner's cost function also for pyATF's methods, ensuring that the lack of a memoization mechanism does not put pyATF's methods at a disadvantage. Moreover, this approach also ensures all optimization algorithms in our comparison use the same backends, compilers, and time measurement method. We compare against all optimization algorithms currently in pyATF; of these, *simulated annealing*, *differential evolution*, *Pattern Search*, and *Torczon* are stand-alone algorithms, while *AUC Bandit* and *Round Robin* are ensemble-based approaches re-using the stand-alone algorithms.

We show the results of this comparison in Fig. 6.6, where it can be seen that of the six pyATF algorithms, only *differential evolution* (abbreviated to *DE*) performs on par with our constraint-aware optimization algorithms, while especially *AUC Bandit*, *Simulated Annealing* (*SA*), *Torzon* and *Pattern Search* perform notably worse than even the non-constrained Kernel Tuner optimization algorithms shown in Fig. 6.4. The mean difference with the baseline Quantifying this with the performance score, the average score of our optimization algorithms is 0.264, as opposed to -2.361 for pyATF.

To validate our findings we can compare the performance score per search space between the pyATF and the constrained-aware optimization algorithms we introduce in this work. This is shown in Fig. 6.7 for *simulated annealing*, which is present in both frame-



(a) pyATF Simulated Annealing

(b) Our constrained SA

(c) pyATF Differential Evolution

(d) Our constrained Firefly

Figure 6.7: Performance difference per search space for *Simulated Annealing* and the two best-performing algorithms between pyATF and the constraint-aware implementations presented in this work.

works, as well as for the best-performing algorithm in pyATF and this work. In the latter case, our constraint-aware Firefly (Fig. 6.7d) outperforms pyATF Differential Evolution (Fig. 6.7c) on the majority of search spaces. The pyATF implementation of *Simulated Annealing* (Fig. 6.7a) is outperformed by our SA implementation (Fig. 6.7b) on all but one of the 24 search spaces. While the improvement over pyATF of our optimization algorithms is not necessarily tied to the search space density for SA (as seen in Table 6.3), we can see that the largest improvements in the performance of our Firefly algorithm over pyATF's DE occur in the most sparse auto-tuning search spaces, +0.793 for Hotspot on A6000 and +0.515 for GEMM on A100. This indicates that the performance differences are not only due to underlying differences between the implementations of the optimization algorithms in both frameworks, but also due to differences in the handling of constraints.

6.6 Conclusion

Automatic performance tuning is essential in high-performance computing for achieving optimal application performance across diverse hardware platforms. However, the presence of complex parameter interdependencies and hardware constraints introduces a significant challenge in the form of constrained optimization. In this work, we addressed this challenge by integrating constraint-handling strategies into four widely-used evolutionary optimization algorithms, Simulated Annealing, Particle Swarm Optimization, Firefly, and Genetic Algorithm, within a generic auto-tuning framework.

Our results demonstrate that equipping these algorithms with constraint-awareness leads to substantial performance improvements compared to their unconstrained counterparts. We observed faster convergence, more efficient exploration of the feasible search space, and higher-quality solutions across a broad range of auto-tuning benchmarks. Furthermore, we showed that our methods outperform the state-of-the-art pyATF framework in several tuning scenarios, especially for heavily constrained problems, underscoring the practical value of our approach.

The constrained optimization algorithms presented in this paper have been made available as open-source contributions to the open-source Kernel Tuner software, lowering the barrier to adoption and enabling reproducibility. Future work may include extending support to other classes of optimization algorithms, exploring adaptive constraint-handling techniques, and applying our methods to soft constraints that arise in multi-objective optimization. For example, to find the fastest configuration within a user-specified energy budget.

7 Automated Algorithm Design for Auto-Tuning Optimizers

> *We shape our tools, and thereafter our tools shape us.*
>
> Marshall McLuhan

This chapter investigates the novel idea of using large language models (LLMs) to automatically generate optimization algorithms for auto-tuning. We combine LLMs with an evolutionary framework, ensuring only effective algorithms survive, and evaluate their performance against widely used human-designed methods across a benchmark suite of real-world GPU applications. Our results show that LLM-generated algorithms can reach, and in various cases exceed, the performance of classical approaches, particularly when given problem-specific information. The best-performing generated optimizers are released as part of the open-source Kernel Tuner framework, offering the community a new direction in automated algorithm design for high-performance computing.

This chapter is based on "Automated Algorithm Design for Auto-Tuning Optimizers" by Willemsen, van Stein, and van Werkhoven [174].

7.1 Introduction

A central challenge in auto-tuning lies in efficiently navigating the large and irregular search spaces, described in Chapter 4, as exhaustive exploration is infeasible [133, 136]. As a result, the effectiveness of an auto-tuner critically depends on its optimization algorithm. Classical approaches, including Simulated Annealing, Genetic Algorithms, and Particle Swarm Optimization, have demonstrated strong performance in this domain, but require careful hyperparameter tuning as conducted in Chapter 5 and are often not designed with the search space characteristics of auto-tuning in mind (see Chapter 2).

Meanwhile, recent advances in large language models (LLMs) have demonstrated remarkable capabilities in code generation, algorithm synthesis, and problem-solving across a wide range of domains [28, 91]. These capabilities raise the question: Can LLMs be leveraged to automatically generate effective optimization algorithms for auto-tuning? Such an approach could enable the rapid design of problem-specific optimizers without the need for expert-crafted heuristics, potentially unlocking new levels of efficiency in auto-tuning.

In this chapter, we explore this novel direction by investigating the use of LLMs to generate optimization algorithms for auto-tuning. We use the LLaMEA framework, which combines LLMs with an evolutionary algorithm to automatically generate metaheuristics [146]. We emphasize that only the optimization algorithm is generated, not the underlying applications to be auto-tuned, nor program variants. Hence, the correctness of the generated code is of lesser concern; if the generated algorithm does not run or optimize correctly, it will not survive the selection phase of the evolutionary algorithm. Specifically, we evaluate whether LLM-generated algorithms can match or exceed the performance of human-designed optimization methods across a diverse benchmark suite, and whether such algorithms benefit from problem-specific information. Our work is, to the best of our knowledge, the first to systematically study the application of automated algorithm design in the context of auto-tuning.

We make the following contributions:

- We introduce a method for leveraging LLMs to automatically generate optimization algorithms tailored for auto-tuning problems.
- We evaluate LLM-generated optimizers against widely used classical algorithms across a representative set of real-world applications and architectures.
- We determine whether generated algorithms benefit from problem-specific information.

- The best-performing generated algorithms are incorporated into the Kernel Tuner framework, benefiting users.
- All code and results are available in our GitHub repository[1].

The remainder of this chapter is structured as follows. Section 7.2 reviews related research on optimization methods and recent advances in LLM-based algorithm synthesis. Section 7.3 details our approach for generating and evaluating optimization algorithms with LLMs. Section 7.4 presents a comprehensive evaluation. Finally, Section 7.5 concludes.

7.2 Related Work

Auto-tuning is established as a key technique for optimizing performance-critical applications on modern heterogeneous systems [9]. ATLAS [168] pioneered empirical auto-tuning by generating and benchmarking parameterized versions of dense linear algebra kernels, achieving performance portability across architectures. FFTW [50] introduced an adaptive planner that automatically searches among composition strategies for fast Fourier transforms. A number of frameworks have been developed over the years to generalize these application-specific approaches to arbitrary applications. For example, Active Harmony [152] and Orio [62] extended auto-tuning to runtime adaptation and annotation-based empirical tuning, respectively. Open Tuner [3] is an extensible framework for application-level auto-tuning. CLTune [114] is a generic auto-tuner for OpenCL kernels. Traditionally, numerical optimization methods and metaheuristics have been widely used in these approaches, including Nelder-Mead, Genetic Algorithms (GA), Simulated Annealing (SA), Particle Swarm Optimization (PSO) [152, 62, 165, 134, 3, 114]. Although effective in many settings, these algorithms generally require many empirical evaluations and careful hyperparameter choices to achieve good performance across diverse auto-tuning tasks, as described in Chapter 5.

7.2.1 Machine Learning and Hybrid approaches

Beyond traditional optimization algorithms, several auto-tuning approaches have explored alternative strategies. Multi-armed bandits and ensemble methods, as used in OpenTuner [3] and ATF [125]/PyATF [135], dynamically allocate budget across different optimizers, leveraging their strengths on different problem classes. Machine learning techniques have also been applied: predictive modeling approaches such as GPTune [95], ytopt [175], and AUMA [47] use regression models or Gaussian processes to guide the search towards

[1] https://anonymous.4open.science/r/BLADE-AutoTuner/README.md

promising configurations. KTT uses a machine-learning-based optimization method that incorporates performance counter data collected by a profiler [48]. Bayesian optimization frameworks specialized for high-performance computing, such as HyperMapper [111] and Chapter 2, further demonstrated the potential of learning-based auto-tuning. These methods show that performance can be improved not only by choosing better traditional optimizers, but also through hybrid, model-based, or domain-specific techniques [35, 176].

7.2.2 Meta-optimization techniques

Recent work has begun to focus on meta-optimization, i.e., improving the auto-tuning process itself. Chapter 4 studied how constraints shape the construction of search spaces in auto-tuning, while Chapter 5 introduced hyperparameter optimization for tuning algorithms within auto-tuners, demonstrating substantial performance gains.

Constraint-aware optimization, as discussed in Chapter 6, further improves efficiency by preventing wasted evaluations on invalid configurations. Similarly, BaCO [68] automatically learns so-called *hidden constraints* to exclude configurations that turn out to be infeasible during compilation or at run time. These contributions highlight the importance of adapting optimization algorithms to the unique challenges of auto-tuning.

7.2.3 Automated Algorithm Design

In parallel, the rise of large language models (LLMs) has inspired their application in code generation, software engineering, and algorithm design. This opened up a new line of research, where LLMs are employed as generative engines for algorithms themselves. FunSearch [129] demonstrated that LLMs can discover competitive search procedures for combinatorial problems by generating functional code that is iteratively refined. Similarly, the Evolution of Heuristics (EoH) framework [94] and ReEvo [179] explore the automated design of heuristics and metaheuristics through evolutionary search guided by LLMs. Furthermore, LLaMEA (Large Language Model Evolutionary Algorithm) [146] automatically evolves complete metaheuristic algorithms, showing that LLMs can propose novel optimization algorithms that are subsequently evaluated and selected in an evolutionary loop. Taken together, these works suggest a paradigm shift where optimizers are no longer hand-crafted but instead automatically generated, adapted, and evolved. An approach particularly promising for auto-tuning, where search spaces are large, irregular, and domain-specific.

While LLMs have been shown to be capable of generating competitive algorithms for classical continuous and combinatorial black-box optimization problems [146, 148], their use in the domain of auto-tuning remains unexplored. Our work is the first to investigate

LLMs as generative components for optimization algorithms within auto-tuning frameworks. Unlike most prior work, we do not rely on human-designed algorithm templates or hyperparameter tuning alone, but instead allow LLMs to propose entirely new optimization strategies, which are then evaluated under the rigorous autotuning methodology described in Chapter 3.

7.3 Design and Implementation

This section describes the design and implementation of our approach. Figure 7.1 shows an overview of the design of our system to automatically generate optimization algorithms for auto-tuning. The LLaMEA search process is guided by an evolutionary algorithm (EA) that acts as a meta-strategy:

1. A population of optimization algorithms is initialized with candidates generated by the LLM. In our case, LLaMEA starts with 4 parent solutions.
2. Each candidate algorithm is evaluated within Kernel Tuner using the autotuning methodology performance score $\mathscr{P}$ on the training set.
3. Algorithms with higher scores are selected for reproduction, while poorly performing ones are discarded.
4. A variety of LLM-driven mutation operators generate new candidate algorithms (in this case 12), including diversity-focused mutation operators.

Steps 2-4 are repeated for a fixed number of generations or until the evaluation budget is exhausted.

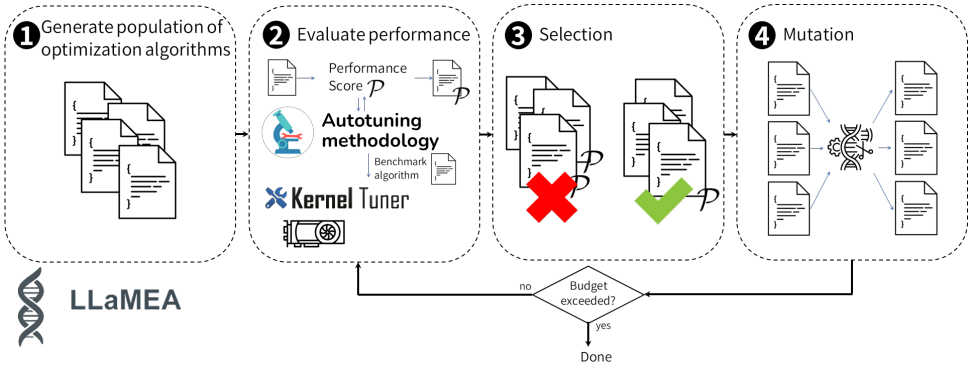


Figure 7.1: Overview of our designed system, showing how LLaMEA generates optimization algorithms, which are evaluated using the autotuning methodology, which tests the performance of the optimization algorithm in Kernel Tuner.

This design ensures that the LLM plays a creative but non-critical role: it proposes optimization algorithms, while the EA-based selection mechanism ensures only high-performing candidates survive. Because evaluation is entirely based on measured performance in Kernel Tuner, the process is robust to errors or inefficiencies in LLM-generated code. In addition, LLaMEA handles time-outs and errors in generated code by feeding back the stack-traces as additional context for the LLM, which allows for self-debugging when needed[2].

The rest of this section explains the individual components and details how these are integrated to work in concert. We first introduce Kernel Tuner as the target auto-tuning framework in Section 7.3.1, followed by Section 7.3.2 with an overview of how we have used the LLaMEA system for LLM-assisted automated algorithm design. Finally, we explain how these components are integrated in Section 7.3.3: LLaMEA generates Kernel Tuner-compatible optimization algorithms that are evaluated with a performance score, while an evolutionary algorithm guides the evolutionary search over candidate algorithms.

7.3.1 Kernel Tuner

Kernel Tuner is a Python-based, open-source auto-tuning framework designed for optimizing computational kernels [165]. It supports multiple programming backends, including CUDA, HIP, OpenCL, and OpenACC, C++ and Fortran, and is widely adopted for tuning applications across different architectures and objectives, such as execution time and energy efficiency.

Fig. 7.2 shows the overview of the software architecture of Kernel Tuner. Users provide Kernel Tuner with a kernel implementation and a Python script that specifies the input data, a description of the tunable parameters, and possibly any constraints. At the start of the tuning process, Kernel Tuner constructs a search space $\mathscr{X}$ consisting of all valid configurations. The optimization algorithm, called an optimization strategy in Kernel Tuner, iteratively selects candidate solutions $x \in \mathscr{X}$ to be compiled, executed, and measured. The evaluation metric, typically kernel runtime, determines the quality of the configuration. Kernel Tuner internally abstracts the different backends into a unified interface, allowing most of the tool to operate independently of the programming language and target hardware platform.

More formally, auto-tuning involves optimizing an application or *kernel* K_i on a target system G_j for input data I_k to maximize performance $f_{G_j,I_k}(K_i)$. The auto-tuner constructs a search space $\mathscr{X}$ by considering all tunable parameters and their valid values, subject to

[2]See https://anonymous.4open.science/r/BLADE-AutoTuner/README.md

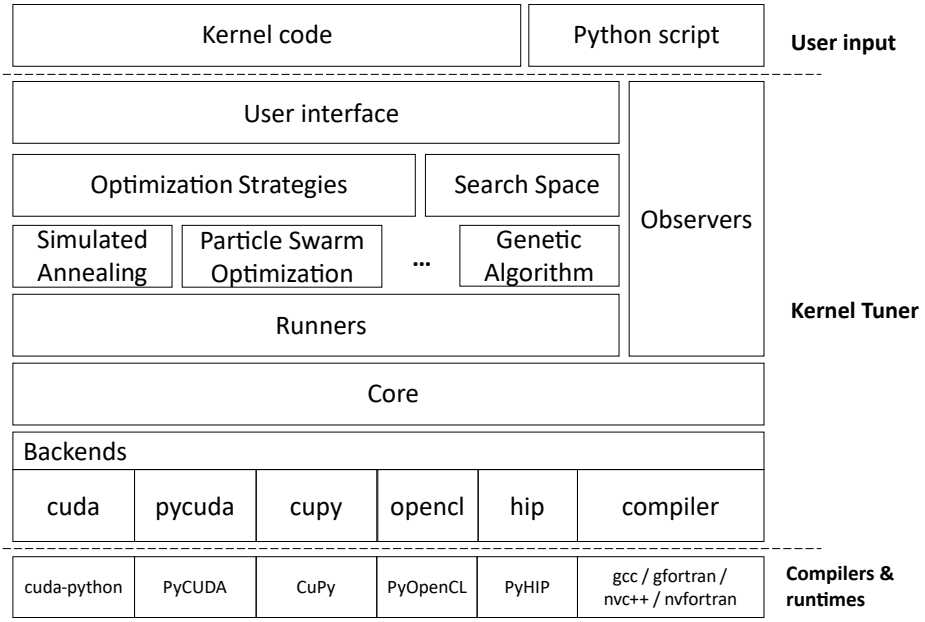


Figure 7.2: Overview of the software architecture of Kernel Tuner.

user-defined constraints, as described in Chapter 4. The objective is to determine the optimal configuration, or more formally (assuming minimization), as follows:

$$x^{\star} = \arg\min_{x \in \mathscr{X}} f_{G_j, I_k}(K_{i,x}). \tag{7.1}$$

The evaluation of each configuration takes time, as it needs to be compiled and executed on the target system. Search spaces in auto-tuning are often large, discontinuous, non-convex, and irregular. These characteristics make them infeasible to search by hand and demand carefully chosen optimization algorithms. If the algorithm is not well adapted to the search space, the number of evaluations taken to find a near-optimal configuration becomes excessive, limiting the practical usability of the auto-tuner. We have extended Kernel Tuner with the capability to use any externally-defined optimization algorithm, not only those already included in the Kernel Tuner source code. To this end, we have implemented an `OptAlg` wrapper class that can be used to wrap optimization algorithms in a format that Kernel Tuner supports.

7.3.2 LLaMEA

LLaMEA (Large Language Model-Assisted Meta-Evolutionary Algorithms) [146] is a system designed to leverage the generative capabilities of large language models for the syn-

thesis of optimization algorithms. In our setting, the scope of the LLM is *strictly limited to generating the optimization algorithm logic*. This ensures that the LLM does not interact with application kernels, data, execution, or numerical accuracy. Consequently, correctness and reproducibility of results are guaranteed, as the kernel execution pipeline remains unchanged.

The LLM generates code snippets that define candidate optimization algorithms. The optimization algorithms specify how new configurations are sampled, selected, and evaluated within Kernel Tuner. If a generated algorithm is incorrect or contains implementation errors, it simply yields poor performance when evaluated, resulting in a low performance score (fitness). These weak algorithms are discarded during the evolutionary search process.

The initial population of candidate optimization algorithms is generated using the prompt template shown in Fig. 7.3. The `code format specification` briefly explains how to use Kernel Tuner's `OptAlg` base class as well as the interface of the `SearchSpace` object in Kernel Tuner (see Fig. 7.2). Optionally, we can insert information about the specific tuning problem at hand to allow the LLM to incorporate information on the tunable parameters, their possible values, and constraints. The `minimum working code example` includes an example implementation of an empty optimization algorithm template, illustrating how to use the `SearchSpace` object to: 1) generate an initial population, 2) retrieve neighbors of a particular configuration, and 3) repair any invalid configuration that violates constraints. Finally, the `output format specification` instructs the LLM to first print a one-line description followed by the code. The complete prompts used can be found in our online repository [3].

LLaMEA employs an iterative evolutionary loop in which LLMs generate candidate metaheuristic algorithms in executable code, which are then autonomously evaluated to provide performance feedback. The framework supports mutation and selection strategies common in evolutionary computing algorithms with different solution population sizes.

Code evolution is implemented via different natural language-based mutation prompts (See Fig. 7.4). These prompts are selected to give both exploration and exploitation behaviour in code space. LLaMEA for Kernel Tuner uses an elitism strategy with 4 parent solutions and 12 offspring solutions per iteration. Beyond automated algorithm discovery, LLaMEA offers capabilities such as in-the-loop hyperparameter tuning [148], where numerical tuning is delegated to hyperparameter optimization tools, allowing the LLM to focus on structural innovation.

[3] https://anonymous.4open.science/r/BLADE-AutoTuner/README.md

Task Prompt

Your task is to design novel metaheuristic algorithms to solve kernel tuner problems (integer, variable dimension, constraint).
<code format specification>
<OPTIONAL search space specification (json)>
An example code structure with helper functions is as follows:
<minimum working code example>
Give an excellent and novel heuristic algorithm to solve this task and also give it a one-line description, describing the main idea.
<output format specification>

Figure 7.3: Prompt template used to generate initial candidate optimization algorithms.

Mutation Prompts

- Refine the strategy of the selected solution to improve it.
- Generate a new algorithm that is different from the algorithms you have tried before.
- Refine and simplify the selected algorithm to improve it.

Figure 7.4: Mutation prompts used by LLaMEA for Kernel Tuner.

7.3.3 Evaluation of Automatically Generated Algorithms

Kernel Tuner includes more than 20 optimization algorithms [134], ranging from local search to population-based methods. However, large and irregular search spaces present in auto-tuning make algorithm design challenging. Ineffective optimizers waste evaluations on poor configurations, reducing tuning efficiency. Our approach addresses this challenge by automatically generating novel optimization algorithms specifically tailored for Kernel Tuner. Nevertheless, evaluating whether an optimization algorithm is actually well-suited for auto-tuning is challenging; as the large, irregular search spaces encountered in auto-tuning can be difficult to optimize, an optimizer that performs well on one search space may perform poorly on another. A robust, systematic approach to evaluation is thus needed to automatically generate suitable optimization algorithms.

In Chapter 3, we presented a community-driven methodology for evaluating optimization algorithms in auto-tuning, which can be used to evaluate LLaMEA algorithms with Kernel Tuner. This methodology provides a systematic approach to comparing optimization algorithms across auto-tuning search spaces. It defines a performance score $\mathscr{P}$ that

quantifies an optimization algorithm's performance over the passed time relative to a calculated baseline, typically random search, to have consistent, objective-independent, transparent, and comparable behavior across search spaces.

On an individual search space, this approach first defines a budget and performance baseline adapted to the search space characteristics. The optimization algorithms to be compared are then executed multiple times to account for noise, after which the difference between the baseline and the average best performance of each optimization algorithm is compared at fixed, equidistant time intervals relative to the budget. The result is a smooth performance curve over time, instead of relying solely on final performance. By adapting the budget and baseline to reliable search space characteristics, we can compare and aggregate these performance curves across search spaces. We will thus use the mean of these aggregated performance curves as a performance score $\mathscr{P}$ of an optimization algorithm, where a higher score is better overall performance.

More formally, for a given time sampling point t the performance $\mathscr{P}_t$ of an optimization algorithm $\mathscr{F}$ can be computed:

$$\mathscr{P}(\mathscr{F}, G_j, K_i, I_k)_t = \frac{\mathscr{S}_{\text{baseline}}(t) - \mathscr{F}(G_j, K_i, I_k)_t}{S_{\text{baseline}}(t) - \mathscr{S}_{\text{opt}}} \tag{7.2}$$

where $\mathscr{S}(G_j, K_i, I_k)$ are the search space characteristics given a target system G_j, kernel K_i, and input data I_k. Here, $\mathscr{F}_t$ is the best objective value found so far by the optimization algorithm, $\mathscr{S}_{\text{baseline}}(t)$ is the performance of the baseline method, and $\mathscr{S}_{\text{opt}}$ is the known optimal value in the search space. This yields $\mathscr{P}_t = 0$ when performance equals the baseline and $\mathscr{P}_t = 1$ when the optimum has been found.

To avoid distorting the metric with large variations in search space difficulty, evaluations are limited to the time at which the baseline reaches a set cutoff percentile between the median and the optimum, typically somewhere around 95%, referred to as the budget. With this method, the performance curves are relative to the same baseline and optimum, the cutoff provides a consistent reference point, and the sampling points are equidistant. The performance curves can thus be meaningfully aggregated from different search spaces by taking the mean score of all curves at each t, resulting in the aggregate performance curve over all search spaces for the optimization algorithm. The aggregate performance score is then obtained by averaging over the set of discrete time sampling points $\mathscr{T}$, or more formally:

$$\mathscr{P}(\mathscr{F}, K, G, I) = \frac{1}{|\mathscr{T}|} \sum_{t \in \mathscr{T}} \frac{\sum_{K_i \in K} \sum_{G_j \in G} \sum_{I_k \in I} \mathscr{P}(\mathscr{F}, G_j, K_i, I_k)_t}{|K||G||I|} \tag{7.3}$$

The kernels K, target systems G, and inputs I are the collections of K_i, G_j, and I_k of Eq. (7.1) on which the LLaMEA algorithms are evaluated, hence referred to as the training set. This aggregate score enables robust comparison of optimization algorithms by capturing both the quality of the configurations found as well as the time taken to do so. As such, a difference when comparing two performance scores can indicate a difference in the quality of configurations found, the time taken to do so, or a combination of both.

7.4 Evaluation

In this section, we evaluate the effectiveness of large language model (LLM)-generated optimization algorithms as introduced in Section 7.3. Section 7.4.1 describes the experimental setup. We then address our two main research questions: whether our LLM-generated algorithms benefit from incorporating additional problem-specific information (Section 7.4.2), and whether our generated algorithms can match or exceed the performance of established optimization methods for auto-tuning (Section 7.4.4). In between, Section 7.4.3 discusses the details of the two best generated optimization algorithms.

7.4.1 Experimental Setup

The experimental setup consists of three sections: Section 7.4.1 detailing the benchmarks evaluated on, Section 7.4.1 describing the hardware used, and an account of the software in Section 7.4.1.

We will evaluate our approach using the BAT benchmark suite [159] of auto-tunable GPU kernels. Specifically, we use the *dedispersion*, *convolution*, *hotspot*, and *GEMM* kernels. These benchmark kernels are examples of widely used real-world applications in astronomy, image processing, materials science, and linear algebra, respectively. The characteristics of these applications are shown in Table 7.1. The Cartesian size is the size of the complete combinatorial space without the application of constraints. The constrained size is the number of feasible solutions that remain after applying constraints. The number of dimensions equals the number of tunable parameters in the source code. We will use

Table 7.1: Overview of the basic characteristics of the real-world applications.

Name	Cartesian size	Constrained size	Dimensions
Dedispersion	22272	11130	8
2D Convolution	10240	4362	10
Hotspot	22200000	349853	11
GEMM	663552	116928	17

these four applications as a benchmark throughout this evaluation. Besides their diverse origins, they also bring diversity in their performance characteristics; e.g., dedispersion and hotspot are generally bandwidth-bound, while convolution and GEMM are generally compute-bound; we will explore these applications in more detail.

The *Dedispersion* kernel in BAT originates from the AMBER pipeline for the detection of single-pulse astronomical transients [137]. Dedispersion is a signal processing operation that compensates for the frequency-dependent time delay experienced by radio waves as they propagate through space. This delay, caused by dispersion in the interstellar medium, results in lower-frequency components of the signal arriving later than higher-frequency components that were emitted simultaneously. For a signal with the highest frequency f_h received at time t_x, the delay k for a lower frequency f_i can be approximated by:

$$k \approx 4150 \times DM \times \left(\frac{1}{f_i^2} - \frac{1}{f_h^2} \right),$$

where DM is the dispersion measure, characterizing the integrated column density of free electrons between the source and the observer.

The kernel takes as input time-domain samples across many frequency channels and outputs dedispersed samples for a range of DM values. It is parallelized such that each thread can process multiple time samples and dispersion measures in parallel, iterating over the frequency bands. For this study, we use parameters based on the ARTS survey on the Apertif telescope [26], which employs a sampling rate of 24.4 kHz, 2048 dispersion measures, and 1536 frequency channels.

Several tunable parameters influence the performance of this kernel on GPUs. These include the dimensions of the thread blocks, the amount of work assigned per thread in both the time and dispersion measure dimensions, and the strategy used to stride through the input samples. The kernel also allows partial loop unrolling over the frequency channels, where a factor of zero delegates this decision to the CUDA compiler. Additionally, the number of thread blocks per streaming multiprocessor can be controlled to optimize resource utilization.

The *Convolution* kernel, developed by Van Werkhoven et al. [166] as an optimized and highly tunable GPU-accelerated library for 2D convolution operations, has become a widely used benchmark in autotuning research [114, 121, 165, 134]. A two-dimensional convolution is a fundamental operation in image processing and computer vision, where a filter mask is applied across an input image to compute a weighted sum of neighboring pixel values for each output element. Given an input image I of size $(w \times h)$ and a convolu-

tion filter F of size $(F_w \times F_h)$, the output image O of size $((w-F_w) \times (h-F_h))$ is computed as:

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

The implementation in BAT exposes several tunable parameters that strongly influence GPU performance. The thread block dimensions in the x and y directions determine the granularity of parallelism and the number of threads per block. Each thread may compute multiple output elements along the x and y axes, depending on the selected tile sizes. A shared memory padding scheme can be enabled to mitigate bank conflicts when block sizes are not multiples of the number of memory banks, as described by Van Werkhoven et al. [166]. Additionally, the kernel can optionally use the read-only cache to reduce global memory traffic when loading input elements.

The *Hotspot* kernel, included in BAT and originating from the Rodinia Benchmark suite [27], is part of a thermal simulation application that estimates the temperature distribution of a processor. The simulation considers the processor's architectural floor plan, thermal resistance, ambient temperature, and simulated power currents. Through an iterative process, the kernel solves a set of differential equations to model heat dissipation over time. The inputs to the kernel are the power consumption and initial temperature values, and the output is a two-dimensional grid of average temperature values across the chip.

The performance of the Hotspot kernel is highly dependent on a number of tunable parameters that influence how work is distributed across GPU threads and how data is managed in memory. The thread block dimensions in the x and y directions define the number of threads per block, with between 32 and 1024 threads typically used. Each thread may compute one or more output elements in each dimension. The temporal tiling factor controls whether multiple iterations of the stencil operation are to be performed within a single kernel launch, improving data locality. Finally, shared memory usage for caching power input data can be enabled or disabled, and another parameter controls the number of thread blocks per SM to influence occupancy and register usage.

Generalized dense matrix-matrix multiplication (GEMM) is a fundamental operation defined in the BLAS linear algebra specification and is extensively used across scientific computing, machine learning, and high-performance computing applications. It is also a common benchmark for GPU code optimization studies [114, 90, 133] due to its computational intensity and sensitivity to hardware-specific optimizations. In this evaluation, we use the GEMM kernel from CLBlast [115], a tunable OpenCL BLAS library.

The GEMM operation performs the multiplication of two matrices, A and B, and optionally adds a scaled version of an existing output matrix C:

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

where α and β are scalar coefficients.

The CLBlast GEMM kernel exposes a range of tunable parameters that affect its performance on different GPU architectures. The tunable parameters control the workload assigned to each thread block in two dimensions, while another two parameters control the dimensions of the thread block itself. Loading matrix tiles into shared memory can be enabled or disabled for both matrix A and B individually. When enabled, two more parameters control shared memory level tile sizes. Finally, two more parameters define the vector widths used for global memory transactions, enabling efficient vectorized loads and stores.

These benchmark applications and their described parameters each form a flexible search space for performance tuning across different GPU architectures.

To obtain a diverse set of real-world auto-tuning cases for evaluation, we use the four auto-tuning applications described in Section 7.4.1 on a diverse set of six GPUs from the DAS-6 [8] and LUMI [81] supercomputers, resulting in 24 unique auto-tuning search spaces. For a fair evaluation, the LLaMEA optimization algorithm feedback loop uses a training set of twelve search spaces resulting from the four applications on the AMD MI250X, Nvidia A100, and Nvidia A4000, and the test set consists of twelve search spaces resulting from the four applications on the AMD W6600, AMD W7800, and Nvidia A6000. These search spaces have been exhaustively explored to allow efficient evaluation using simulation, as described in Chapter 5 and provided for public use by the authors. The evaluation of this work is conducted on the A4000 nodes of DAS6. The nodes in DAS-6 have a 24-core AMD EPYC-2 7402P CPU and 128 GB of RAM.

The OS used is Rocky Linux 8 with Linux kernel version 4.18, the Python version 3.11.7, as well as Kernel Tuner 1.3.0, and PyATF 0.0.9. The version of LLaMEA used is 1.0.6, and GPT o4-mini with the default hyperparameters is used as the LLM.

For performance comparison, we follow the auto-tuning methodology community standard of Chapter 3, as discussed in Section 7.3.3. Each optimization algorithm is evaluated using a *performance score* $\mathscr{P}$ that measures the quality of configurations found over

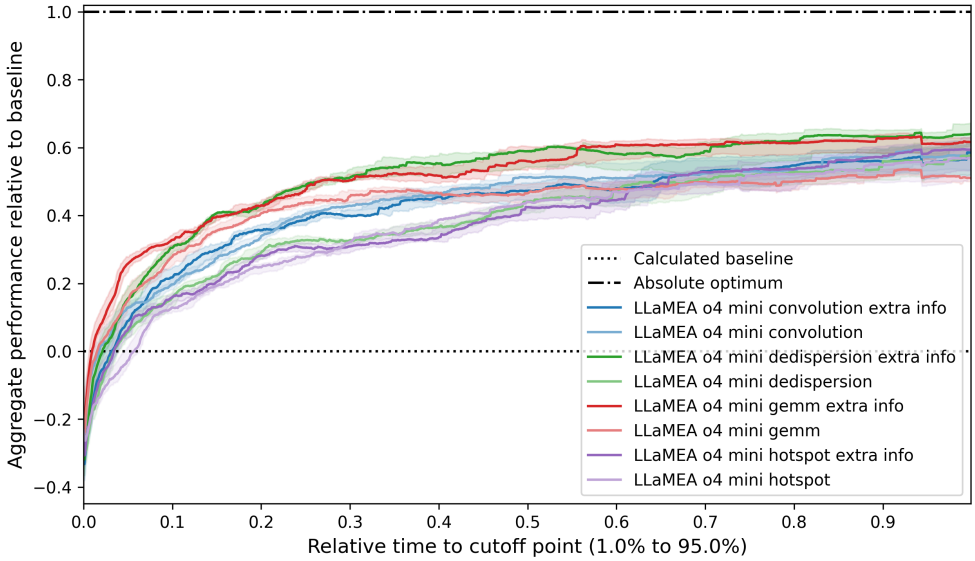


Figure 7.5: The aggregate performance over time over all search spaces of the application-specific optimization algorithms, each with and without additional search space knowledge. Mean of 100 runs each, the shaded area marks the 95% confidence interval.

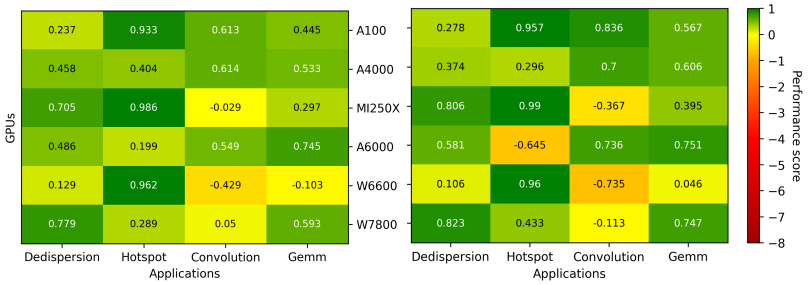
time relative to a statistically estimated *random search baseline*. This aggregate metric captures both search efficiency and solution quality across different search spaces. A score of 0.0 indicates parity with the baseline, while a score of 1.0 corresponds to consistently finding the global optimum immediately. The optimization budget per run is defined as the time required by the baseline to reach 95% of the distance between the search space median and optimum. Each algorithm variant is executed 100 times per search space to mitigate stochasticity.

7.4.2 Impact of Target Application and Additional Information

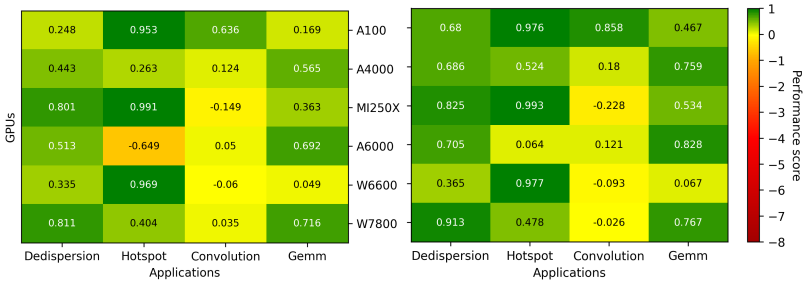
For each application, we generated two algorithms using the LLaMEA-Kernel Tuner loop of Section 7.3:

1. **Without search space knowledge**: provided solely with a hand-crafted auto-tuning task description.
2. **With search space knowledge**: provided with information on possible parameter values and constraints.

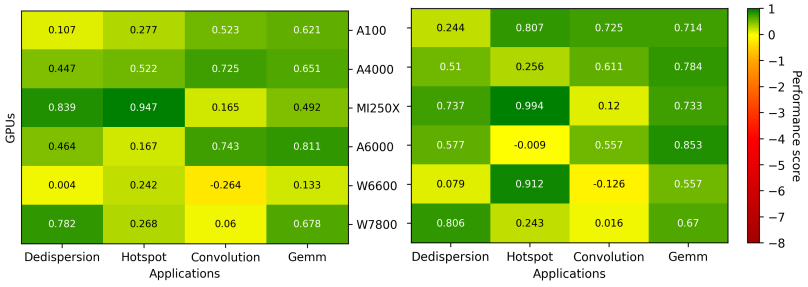
Starting with the overall performance of both versions generated for each application across all search spaces in Fig. 7.5, it is remarkable that all final versions of the generated optimization algorithms perform well in general. In these aggregate plots, each line



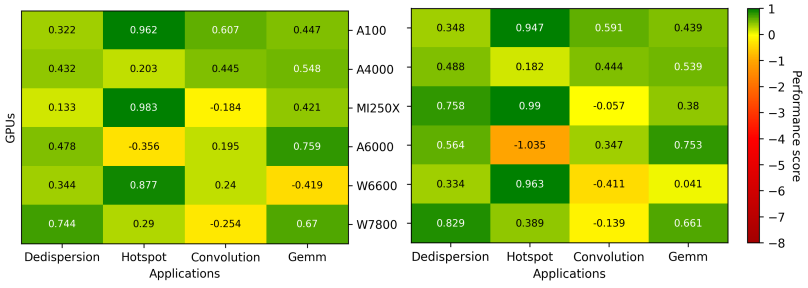
(a) Convolution **(b)** Convolution with extra info



(c) Dedispersion **(d)** Dedispersion with extra info



(e) GEMM **(f)** GEMM with extra info



(g) Hotspot **(h)** Hotspot with extra info

Figure 7.6: Optimization algorithm performance per search space.

Table 7.2: Overall performance scores and standard deviations of both algorithm versions for each target application.

Target application	Without extra info	With extra info	Difference
Convolution	0.429 *0.152*	0.426 *0.157*	-0.003
Dedispersion	0.383 *0.162*	0.515 *0.164*	+0.132
GEMM	0.432 *0.119*	0.516 *0.136*	+0.084
Hotspot	0.373 *0.177*	0.388 *0.172*	+0.015
Mean	0.404	0.463	+0.057

Table 7.3: Overall performance scores of non-target against target algorithms on the applications.

Target application	Non-target mean score	Target score	Difference
Convolution without extra info	0.195	0.219	+0.024
Convolution with extra info	0.195	0.176	-0.019
Dedispersion without extra info	0.476	0.526	+0.050
Dedispersion with extra info	0.476	0.695	+0.219
GEMM without extra info	0.468	0.564	+0.096
GEMM with extra info	0.468	0.719	+0.251
Hotspot without extra info	0.538	0.493	-0.045
Hotspot with extra info	0.538	0.404	-0.134
Mean	0.419	0.475	+0.055

denotes the mean performance over time of the 100 runs, and the shaded area its 95% confidence interval. The performance scores are the means of this performance over time. As seen in Table 7.2, the additional information on the search space provided in the prompt appears to have had an overall positive influence, substantially so for the algorithms generated with target applications *dedispersion* and *GEMM*. On average, providing the additional information increased the performance score by 14.6%.

To further investigate these findings, we can compare the performance score per search space between the algorithms that did and did not receive additional search space information, shown in Fig. 7.6. This shows that while there is some correlation between the target-application and performance, in general, the algorithms appear to generalize well, both outside their target application and outside the set of GPUs trained on.

On a per-application level, we can distinguish between cases where an algorithm was targeted towards an application and where it was not. This is shown in Table 7.3, where the *non-target mean score* denotes the average performance score for that application of the algorithms not targeted towards it, which is compared to the two algorithms that were targeted for each application. While the two *hotspot*-targeted algorithms and the *convolution*-targeted algorithm with extra information perform worse than their non-targeted

Algorithm 6: HybridVNDX high-level pseudocode

Input : Objective f, search space $\mathscr{X}$ with neighbourhoods; budget via `f.budget_spent_fraction`
Output : Best configuration $x^\star$
Initialize $x \leftarrow$ `random_valid()`, $f_x \leftarrow f(x)$; maintain history $\mathscr{H}$, elite heap $\mathscr{E}$, tabu deque $\mathscr{T}$
Initialize neighbourhood weights $w[\cdot] \leftarrow 1$; temperature $T \leftarrow T_0$
while `f.budget_spent_fraction` < 1 **do**
  Sample neighbourhood N by roulette over w
  Build candidate pool: subset of $N(x)$, 1 elite-crossover child, fill with random valid samples; repair infeasible
  Score each candidate c by k-NN prediction on $\mathscr{H}$ (Hamming), add tabu penalty; pick $\tilde{c} = \arg\min$ score
  Evaluate $f_{\tilde{c}} \leftarrow f(\tilde{c})$; push $(\tilde{c}, f_{\tilde{c}})$ to $\mathscr{H}$ and $\mathscr{E}$;
  if $f_{\tilde{c}} \leq f_x$ ***or*** *rand*() $< \exp(-(f_{\tilde{c}} - f_x)/T)$ **then** $x \leftarrow \tilde{c}$; $f_x \leftarrow f_{\tilde{c}}$; push x to $\mathscr{T}$; $w[N] \leftarrow 1.1\,w[N]$
  else $w[N] \leftarrow 0.9\,w[N]$
  $T \leftarrow \alpha T$ (cooling);
  if *stagnation* $>$ *threshold* **then**
    restart from new random valid x and reset T
Return the best-so-far in $\mathscr{H}$

counterparts, the other five algorithms appear to have benefited substantially from the application-targeted approach, with an average improvement of 30.7%.

7.4.3 Algorithmic Details of the Two Best Generated Optimizers

As reported in Section 7.4.2, enriching the prompt with search-space information particularly benefited the algorithms targeted at *dedispersion* and *GEMM*. To better understand the generated algorithms, we look at the algorithmic details and their approach to auto-tuning problems.

The first algorithm, *HybridVNDX*, combines Variable Neighborhood Descent (VND) [46] with (i) dynamic neighborhood weighting, (ii) a light k-NN surrogate for pre-screening, (iii) elite recombination, and (iv) tabu search [55] and simulated-annealing [161]. The design aims to *quickly* filter promising neighbours per iteration while maintaining diversification and robust escape from local minima.

The default hyperparameters as used in this chapter are: k=5, pool size = 8, restart after 100 non-improving steps, tabu size = 300, elite size = 5, T_0=1.0, cooling = 0.995.

Algorithm 7: AdaptiveTabuGreyWolf

Input : f, $\mathscr{X}$ with $v(\cdot)$ and $N_m(\cdot)$, budget B, hyperparameters $(p, L, s, q, \tau, \rho, T_0, \lambda, T_{\min})$
Output: $x^\star$

$P \leftarrow p$ random valid configs; evaluate; $\mathscr{T} \leftarrow$ recent list of size L;
$x^\star \leftarrow \arg\min_{x \in P} f(x)$; $b \leftarrow 0$
while $b < 1$ **do**
  sort P by f; set leaders (α, β, δ) to the best three;
  foreach $x \in P \setminus \{\alpha, \beta, \delta\}$ **do**
    `// Leader-mixed proposal`
    $y_j \leftarrow \text{Uniform}\{\alpha_j, \beta_j, \delta_j, x_j\}$ for each j;
    `// Shaking`
    with prob. s: **if** *rand()* $< q$ **then** random-dim jump from random valid sample
    else $y \leftarrow$ one-step move in $N_{m(b)}(y)$

    `// Repair, tabu`
    if $\neg v(y)$ **then**
      attempt repair via neighbors; else resample random valid

    if $y \in \mathscr{T}$ **then**
      resample (small Hamming change or fresh sample)

    `// Evaluate and accept`
    Evaluate $f(y)$; $\Delta \leftarrow f(y) - f(x)$; $T \leftarrow \max(T_{\min}, T_0 e^{-\lambda b})$
    if $\Delta \leq 0$ ***or*** *rand()* $< e^{-\Delta/T}$ **then**
      $x \leftarrow y$; push x to $\mathscr{T}$

  update $x^\star$ and stagnation counter; **if** *stagnation* $\geq \tau$ **then**
    reinit worst ρp individuals

  update budget fraction b;
return $x^\star$

The second algorithm, *AdaptiveTabuGreyWolf*, keeps a small population of valid configurations and, at each step, proposes new candidates for every non-leader by mixing each parameter independently from the three current best solutions or the individual itself. A light "shaking" step then perturbs the proposal either by a random coordinate jump from a fresh valid sample or by a one-step move in a discrete neighborhood (using coarser adjacent moves early and stricter ones later) to balance exploration and exploitation. Infeasible proposals are repaired; recently evaluated points are blocked by a tabu list to avoid

repeats. Candidates are accepted with simulated-annealing criteria under a temperature that decays with budget (with mild reheating on stagnation), and if progress stalls, a fraction of the worst population members is randomly reinitialized. The run stops when the evaluation budget is exhausted and the best-so-far configuration is returned. While this algorithm is named differently, after close inspection, a lot of the used elements are similar to the first (e.g., the tabu list and simulated annealing approach).

The default hyperparameters as used in this chapter are: Population size p=8, tabu length L=3p, shake rate s=0.2, jump rate q=0.15, stagnation limit τ=80, restart ratio ρ=0.3, T_0=1.0, λ=5.0, $T_{\min}$=10^{-4}.

These two variants provided the best aggregate performance across the 24 search spaces. To better understand what optimization algorithms perform well on auto-tuning problems, we can conclude that they share two important characteristics: (1) Both methods treat evaluation time as dominant; their additional control logic is lightweight. (2) The neighbour APIs allow architecture-agnostic moves while honouring constraints, which is crucial for large irregular search spaces.

7.4.4 Comparison to Human-designed Auto-Tuning Algorithms

Having evaluated the quality and details of generated optimization algorithms in Section 7.4.2, we now evaluate how they hold up against human-designed auto-tuning optimization algorithms. We compare the best two of our generated algorithms, those targeted towards *dedispersion* and *GEMM* with extra information discussed in Section 7.4.3, against the two best human-designed auto-tuning methods in Kernel Tuner evaluated in Chapter 5. These are Genetic Algorithm (GA), a population-based evolutionary method, and Simulated Annealing (SA), a local search technique with probabilistic acceptance. Both algorithms have undergone extensive hyperparameter tuning for 7 days on the same search spaces as used during the generation stage in this work, under the same conditions, further specified in Chapter 5. To further expand the context of this evaluation, we also compare against the best-performing optimization algorithm from another auto-tuning framework, pyATF [135], which uses a Differential Evolution (DE) implementation. Hyperparameter tuning of pyATF optimizers is not possible without changing the source code (see Chapter 5). The pyATF version used is 0.0.9.

Figure 7.7 shows the aggregate performance of the two LLM-generated algorithms and the three human-designed optimization algorithms across all 24 search spaces. The results indicate that our LLM-generated algorithms achieve more than competitive performance as they outperform the human-designed optimization algorithms by a substantial margin, improving the performance score by 0.126 and 0.282 over Kernel Tuner's GA and

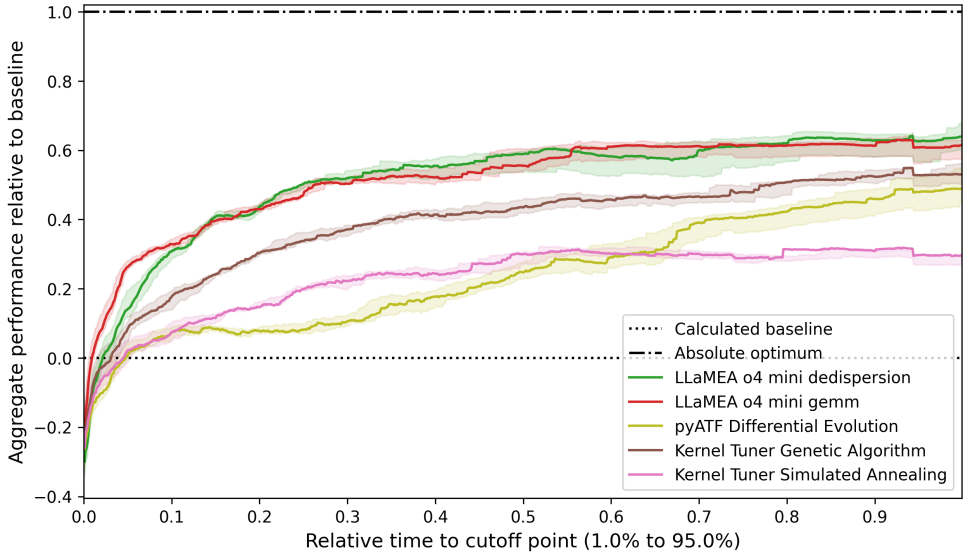
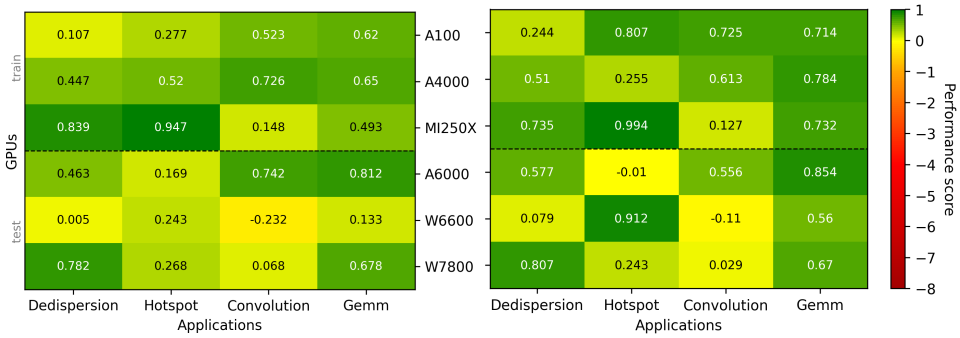


Figure 7.7: The aggregate performance over time over all search spaces of our generated algorithms against three human-designed optimization algorithms. Mean of 100 runs each, the shaded area marks the 95% confidence interval.

SA, respectively. Over pyATF's DE, the performance score is improved by 0.274. On average, our LLM-generated algorithms perform 72.4% better than the human-designed algorithms compared to across the board.

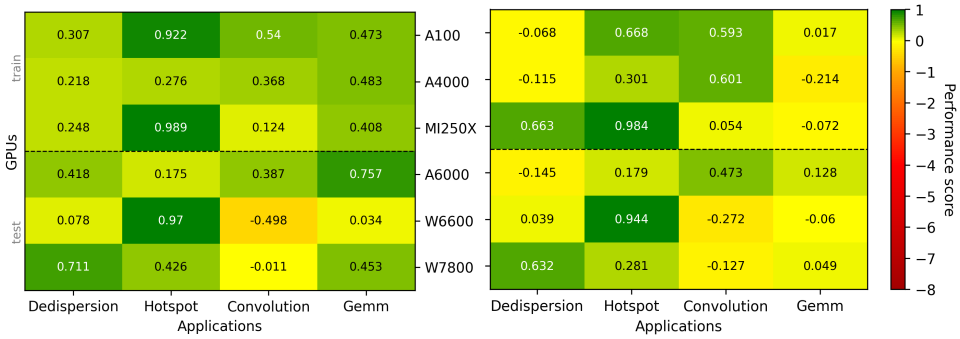
Performance stability across search spaces is also notable, as seen in Fig. 7.8. Comparing the performance of our generated algorithms against the best human-designed algorithm in the evaluation, GA, it is notable that the latter performs better on the *hotspot* application, but is otherwise outperformed in general.

Overall, these results demonstrate that while LLMs can already synthesize competitive optimization algorithms in a general setting, providing additional problem-specific information further boosts their performance, enabling them to surpass human-designed optimization methods.



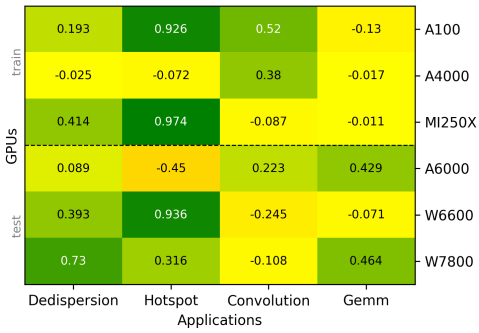
(a) Dedispersion with extra info

(b) GEMM with extra info



(c) Kernel Tuner GA

(d) Kernel Tuner SA



(e) pyATF DE

Figure 7.8: Optimization algorithm performance per search space.

7.5 Conclusion

As modern architectures continue to grow in complexity, auto-tuning remains indispensable for achieving high performance. At the heart of auto-tuning lies the choice of optimization algorithm, which governs how efficiently vast and irregular search spaces can be explored. In this work, we investigated a novel approach: leveraging large language models (LLMs) to automatically generate optimization algorithms tailored to auto-tuning problems.

Our results show that LLM-generated optimization algorithms can achieve performance competitive with, and in various cases superior to, established human-designed algorithms. We observed that these auto-generated methods adapt well to diverse tuning spaces, demonstrating both strong search efficiency and the ability to discover high-performing configurations with limited evaluations. This approach opens new avenues for rapid design and exploration of optimization strategies without requiring extensive manual algorithm engineering.

While our approach shows promise, several limitations should be considered. First, the generated algorithms and their performance depend on the choice of the underlying LLM. We experimented with multiple models, including `o4-mini` and `gemini-2.0-flash`, as well as varying hyperparameters of the LLaMEA framework used for algorithm generation. Although `gemini-2.0-flash` produced valid algorithms, their performance was generally inferior to those generated by `o4-mini`. We selected `o4-mini` for the presented experiments because it offered a favorable balance between performance and computational cost. Another limitation is that our study relies on a fixed set of applications and GPUs to evaluate generated algorithms. Although we demonstrate that LLM-generated algorithms can generalize to a diverse set of unseen search spaces, broader validation across additional architectures and domains can provide further details on their robustness. Finally, while our method currently has a generation-evaluation loop that takes too long to do this, interesting future work could involve direct inclusion in the auto-tuning process by receiving an auto-tuning problem and generating a well-performing optimization algorithm specific to that problem.

In conclusion, our study demonstrates the potential of LLMs to reshape the landscape of auto-tuning by automating the design of optimization algorithms themselves. This paradigm shift not only reduces the human effort required to craft effective search strategies but also paves the way toward more flexible, efficient, and accessible auto-tuning for the next generation of high-performance computing systems. The best-performing optimization algorithms in this work are available in Kernel Tuner, which can be installed with `pip install kernel_tuner`. LLaMEA can be installed with `pip install`

`llamea`. The complete experiment is implemented using the BLADE [147] benchmarking suite, and can be found in our Github repository[4]. For more information, please visit the Kernel Tuner[5] and LLaMEA[6] repositories.

[4] https://anonymous.4open.science/r/BLADE-AutoTuner/README.md
[5] https://github.com/KernelTuner/kernel_tuner
[6] https://github.com/XAI-liacs/LLaMEA

8 Conclusions and Outlook

This thesis has explored how auto-tuning can be made more efficient, scalable, and robust for heterogeneous High-Performance Computing (HPC), with a particular focus on GPU kernel tuning. By addressing challenges in search strategies, evaluation methodology, search space construction, hyperparameter tuning, constraint handling, and automated algorithm design, the work advances the state of the art in performance optimization for modern architectures through six distinct contributions.

Chapter 2 introduced a Bayesian Optimization framework adapted to the discrete and constrained nature of GPU auto-tuning. Through domain-specific design choices such as tailored kernel functions, robust exploration factors, and scalable acquisition functions, the framework consistently outperformed existing strategies. With this, **RQ1** on the suitability of Bayesian Optimization for GPU auto-tuning was answered positively, demonstrating that carefully designed, problem-aware methods outperform generic approaches.

Chapter 3 established a statistically sound methodology for evaluating optimization algorithms. By grounding comparisons in actual time spent and providing an implementation-independent baseline, the framework enables fair and reproducible evaluation. The accompanying open-source package encourages adoption by the community and beyond GPU auto-tuning, for instance, in broader algorithm configuration research. Thus, **RQ2** on how to fairly compare optimization algorithms was addressed, showing that reproducibility and methodological rigor are essential for meaningful progress in the field.

Chapter 4 developed a high-performance Constraint Satisfaction Problem solver to construct large, structured search spaces. Capable of handling billions of configurations in sub-second time, it removes a major bottleneck in auto-tuning workflows. Its integration into open-source tools ensures practical impact. This provides a clear answer to **RQ3** on how to efficiently construct and explore large, constrained search spaces, showing that search space generation no longer needs to be a limiting factor.

Chapter 5 focused on hyperparameter optimization for auto-tuners themselves. By enabling cost-effective large-scale experimentation in simulation mode and creating a FAIR

dataset, performance improvements exceeding 200% are achieved while substantially reducing the energy consumption and resource demand associated with hyperparameter tuning. These results underline the importance of systematic tuning not just for applications, but also for the optimizers that drive auto-tuning. This answers **RQ4**, showing that hyperparameter tuning of the tuner itself can yield substantial improvements in both effectiveness and sustainability.

Chapter 6 extended four evolutionary algorithms with constraint-awareness, improving their performance on the heavily constrained search spaces typical in GPU tuning. These results emphasize that constraints should be treated as first-class citizens in optimization. Consequently, **RQ5** on the impact of constraint handling in evolutionary algorithms was answered affirmatively, as constraint-awareness was shown to be crucial for efficiency and solution quality.

Chapter 7 investigated the automatic generation of optimization algorithms using Large Language Models. By embedding LLMs in a feedback loop with Kernel Tuner, it was shown that bespoke, problem-specific algorithms can outperform even hyperparameter-tuned human-designed methods by wide margins. Although still in an exploratory stage, this line of research suggests a path toward self-improving optimizers capable of adjusting to any problem provided by the user. Hence, **RQ6** was answered by demonstrating that automated algorithm design using LLMs is not only feasible but already competitive, with the potential to fundamentally reshape how optimizers are developed.

Taken together, the thesis demonstrates that progress in auto-tuning is not just made by exploring search spaces more exhaustively, but by improving the tuner itself. By making optimization algorithms domain-aware, rigorously evaluated, efficiently parameterized, constraint-conscious, and even automatically designed, substantial performance and usability gains are achieved, and in doing so, the work answers the overarching research question. The contributions show that careful design, such as by using methods with a robust mathematical foundation, fast and well-defined constraint handling, and hyperparameter tuning, produces optimizers that are tailored to the challenges of GPU auto-tuning. Robust evaluation methodologies ensure that progress is measurable, comparable, and reproducible across diverse problem settings. Continuous enhancement of optimizers, whether by fine-tuning their parameters or by generating entirely new algorithms with LLMs, demonstrates that auto-tuning systems can evolve alongside the hardware and applications they serve.

All methods and tools developed in this thesis are released as open-source software, ensuring reproducibility and enabling continued progress in the field. The advances described in this work enable entirely new problem scales and types to be tackled, not only

in auto-tuning research, but also in HPC applications at large. This includes domains such as climate modeling, astronomy, chemistry, and artificial intelligence, where performance gains and reduced energy use directly benefit both science and society. As heterogeneous HPC systems become ever more central to scientific and industrial computing, the approaches developed here lay the foundations for auto-tuning that is more adaptive, transparent, sustainable, and scalable.

Looking forward, the findings open several promising directions. The methodology, standards, and resources provided in this thesis are already being adopted by the auto-tuning community, providing a shared basis of analysis that enables comparison and reproducibility, thereby fostering scientific growth. The seeding of optimization algorithms and initial sampling methods with prior knowledge of auto-tuning problems using transfer learning could further accelerate the tuning process. Multi-objective optimization and optimization toward energy-performance trade-offs offer natural next steps with substantial practical impact. Finally, automated algorithm design, while still in its infancy, suggests the possibility of self-adapting optimizers that evolve alongside the systems and applications they target. Together, these directions point toward a future where auto-tuning not only meets the growing demands of high-performance computing but also contributes to more sustainable and impactful scientific and industrial computing.

Bibliography

[1] M. Abramowitz and I. Stegun. "Bessel Functions of Fractional Order". In: *Handbook of Mathematical Functions: With Formulas, Graphs, and Mathematical Tables*. Applied mathematics series. Dover Publications, 1965, pp. 435–478.

[2] Z. Alomari, O. E. Halimi, K. Sivaprasad, and C. Pandit. *Comparative studies of six programming languages*. 2015.

[3] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, et al. "Opentuner: An extensible framework for program autotuning". In: *Proceedings of the 23rd international conference on Parallel architectures and compilation*. 2014.

[4] A. H. Ashouri, W. Killian, J. Cavazos, G. Palermo, and C. Silvano. "A Survey on Compiler Autotuning using Machine Learning". In: *ACM Computing Surveys* 51.5 (Sept. 30, 2019), pp. 1–42.

[5] M. Ateeq, H. Habib, A. Umer, and M. U. Rehman. "C++ or python? Which one to begin with: a learner's perspective". In: *2014 international conference on teaching and learning in computing and engineering*. IEEEXplore, 2014, pp. 64–69.

[6] D. Bailey. "Twelve Ways to Fool the Masses When Giving Performance Results on Parallel Computers". In: *Supercomputing Review* 4 (June 9, 1991).

[7] D. H. Bailey. "Misleading performance claims in parallel computations". In: *Proceedings of the 46th Annual Design Automation Conference*. DAC '09. New York, NY, USA: Association for Computing Machinery, July 26, 2009, pp. 528–533.

[8] H. Bal, D. Epema, C. d. Laat, R. v. Nieuwpoort, J. Romein, F. Seinstra, et al. "A Medium-Scale Distributed System for Computer Science Research: Infrastructure for the Long Term". In: *Computer* 49.5 (2016), pp. 54–63.

[9] P. Balaprakash, J. Dongarra, T. Gamblin, M. Hall, J. K. Hollingsworth, B. Norris, et al. "Autotuning in High-Performance Computing Applications". In: *Proceedings of the IEEE* 106.11 (Nov. 2018), pp. 2068–2083.

[10] P. Balaprakash, S. M. Wild, and P. D. Hovland. "Can search algorithms save large-scale automatic performance tuning?" In: *Procedia Computer Science* 4 (2011), pp. 2136–2145.

[11] N. Bannur, H. Maheshwari, S. Jain, S. Shetty, S. Merugu, and A. Raval. "Adaptive COVID-19 Forecasting via Bayesian Optimization". In: *8th ACM IKDD CODS and 26th COMAD*. CODS COMAD 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 432.

[12] R. Bardenet, M. Brendel, B. Kégl, and M. Sebag. "Collaborative hyperparameter tuning". In: *International conference on machine learning*. PMLR, 2013, pp. 199–207.

[13] R. E. Barlow and H. D. Brunk. "The Isotonic Regression Problem and its Dual". In: *Journal of the American Statistical Association* 67.337 (Mar. 1, 1972), pp. 140–147.

[14] C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli. "Satisfiability Modulo Theories. Frontiers in artificial intelligence and applications, vol. 185, ch. 26". In: *Handbook of Satisfiability*. Vol. 185. Frontiers in Artificial Intelligence and Applications. IOS Press, 2008, pp. 825–885.

[15] T. Bartz-Beielstein, C. Doerr, D. v. d. Berg, J. Bossek, S. Chandrasekaran, T. Eftimov, et al. "Benchmarking in Optimization: Best Practice and Open Issues". In: *arXiv:2007.03488 [cs, math, stat]* (Dec. 16, 2020).

[16] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn, and K. Smith. "Cython: The Best of Both Worlds". In: *Computing in Science & Engineering* 13.2 (Mar. 2011), pp. 31–39.

[17] V. Beiranvand, W. Hare, and Y. Lucet. "Best practices for comparing optimization algorithms". In: *Optimization and Engineering* 18.4 (Dec. 1, 2017), pp. 815–848.

[18] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl. "Algorithms for Hyper-Parameter Optimization". In: *Advances in Neural Information Processing Systems*. NIPS'11. Red Hook, NY, USA: Curran Associates Inc., 2011, pp. 2546–2554.

[19] J. Bergstra, N. Pinto, and D. Cox. "Machine learning for predictive auto-tuning with boosted regression trees". In: *2012 Innovative Parallel Computing (InPar)*. 2012 Innovative Parallel Computing (InPar). May 2012, pp. 1–9.

[20] A. Biere, M. Heule, H. van Maaren, and T. Walsh. *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*. NLD: IOS Press, Jan. 2009. 980 pp.

[21] S. C. Billups, S. P. Dirkse, and M. C. Ferris. "A Comparison of Large Scale Mixed Complementarity Problem Solvers". In: *Computational Issues in High Performance Software for Nonlinear Optimization*. Ed. by A. Murli and G. Toraldo. Boston, MA: Springer US, 1997, pp. 3–25.

[22] N. Bjørner, L. de Moura, L. Nachmanson, and C. M. Wintersteiger. "Programming Z3". In: *Engineering Trustworthy Software Systems: 4th International School, SETSS 2018, Chongqing, China, April 7–12, 2018, Tutorial Lectures*. Ed. by J. P. Bowen, Z. Liu, and Z. Zhang. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2019, pp. 148–201.

[23] S. C. Brailsford, C. N. Potts, and B. M. Smith. "Constraint satisfaction problems: Algorithms and applications". In: *European Journal of Operational Research* 119.3 (Dec. 16, 1999), pp. 557–581.

[24] E. Brochu, V. Cora, and N. Freitas. "A Tutorial on Bayesian Optimization of Expensive Cost Functions, with Application to Active User Modeling and Hierarchical Reinforcement Learning". In: *CoRR* abs/1012.2599 (2010).

[25] J. Brownlee. *A note on research methodology and benchmarking optimization algorithms*. Technical Report 70125. Victoria, Australia: Complex Intelligent Systems Laboratory (CIS), Centre for Information Technology Research (CITR), Faculty of Information and Communication Technologies (ICT), Swinburne University of Technology, 2007.

[26] W. van Cappellen, T. Oosterloo, M. Verheijen, E. Adams, B. Adebahr, R. Braun, et al. "Apertif: Phased array feeds for the westerbork synthesis radio telescope-system overview and performance characteristics". In: *Astronomy & Astrophysics* 658 (2022), A146.

[27] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, et al. "Rodinia: A benchmark suite for heterogeneous computing". In: *2009 IEEE International Symposium on Workload Characterization (IISWC)*. 2009 IEEE International Symposium on Workload Characterization (IISWC). Oct. 2009, pp. 44–54.

[28] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, et al. *Evaluating large language models trained on code*. 2021.

[29] R. S. Chen and J. K. Hollingsworth. "ANGEL: A Hierarchical Approach to Multi-Objective Online Auto-Tuning". In: *Proceedings of the 5th International Workshop on Runtime and Operating Systems for Supercomputers - ROSS '15*. the 5th International Workshop. Portland, OR, USA: ACM Press, 2015, pp. 1–8.

[30] M. Cianfriglia, F. Vella, C. Nugteren, A. Lokhmotov, and G. Fursin. *A model-driven approach for a new generation of adaptive libraries*. 2018.

[31] C. A. Coello Coello and E. M. Montes. "Constraint-handling in genetic algorithms through the use of dominance-based tournament selection". In: *Advanced Engineering Informatics* 16.3 (2002), pp. 193–203.

[32] C. A. Coello Coello. "Theoretical and numerical constraint-handling techniques used with evolutionary algorithms: a survey of the state of the art". In: *Computer Methods in Applied Mechanics and Engineering* 191.11 (2002), pp. 1245–1287.

[33] C. A. Coello Coello. "Use of a self-adaptive penalty approach for engineering optimization problems". In: *Computers in Industry* 41.2 (2000), pp. 113–127.

[34] H. Crowder, R. S. Dembo, and J. M. Mulvey. "On Reporting Computational Experiments with Mathematical Software". In: *ACM Transactions on Mathematical Software* 5.2 (June 1979), pp. 193–203.

[35] P. Czarnul and P. Rościszewski. "Auto-tuning methodology for configuration and application parameters of hybrid CPU + GPU parallel systems based on expert knowledge". In: *2019 International Conference on High Performance Computing & Simulation (HPCS)*. 2019 International Conference on High Performance Computing & Simulation (HPCS). July 2019, pp. 551–558.

[36] V. Dalibard, M. Schaarschmidt, and E. Yoneki. "BOAT: Building Auto-Tuners with Structured Bayesian Optimization". In: *Proceedings of the 26th International Conference on World Wide Web*. WWW '17. Republic and Canton of Geneva, CHE: International World Wide Web Conferences Steering Committee, 2017, pp. 479–488.

[37] U. Dastgeer, L. Li, and C. Kessler. "Adaptive Implementation Selection in the SkePU Skeleton Programming Library". In: *Advanced Parallel Processing Technologies*. Ed. by C. Wu and A. Cohen. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2013, pp. 170–183.

[38] R. I. Davis. "On the Evaluation of Schedulability Tests for Real-Time Scheduling Algorithms". In: *7th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*. 2016, p. 9.

[39] E. Daxberger, A. Makarova, M. Turchetta, and A. Krause. "Mixed-Variable Bayesian Optimization". In: *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence (IJCAI-20)* (2020), pp. 2633–2639.

[40] L. De Moura and N. Bjørner. "Z3: An efficient SMT solver". In: *International conference on tools and algorithms for the construction and analysis of systems*. Springer, 2008, pp. 337–340.

[41] K. Deb. "An efficient constraint handling method for genetic algorithms". In: *Computer Methods in Applied Mechanics and Engineering* 186.2 (2000), pp. 311–338.

[42] C. Doerr, H. Wang, F. Ye, S. van Rijn, and T. Bäck. "IOHprofiler: A benchmarking and profiling tool for iterative optimization heuristics". In: *arXiv e-prints:1810.05281* (Oct. 2018).

[43] E. D. Dolan and J. J. Moré. "Benchmarking Optimization Software with Performance Profiles". In: *Mathematical Programming* 91.2 (Jan. 1, 2002), pp. 201–213.

[44] T. Dong, V. Dobrev, T. Kolev, R. Rieben, S. Tomov, and J. Dongarra. "A step towards energy efficient computing: Redesigning a hydrodynamic application on CPU-GPU". In: *2014 IEEE 28th international parallel and distributed processing symposium*. IEEE, 2014, pp. 972–981.

[45] J. Dorn and M. Girsch. "Genetic operators based on constraint repair". In: *Proceedings of the ECAI*. Vol. 94. Citeseer, 1994.

[46] A. Duarte, J. Sánchez-Oro, N. Mladenović, and R. Todosijević. "Variable neighborhood descent". In: *Handbook of heuristics*. Ed. by R. Martí, P. M. Pardalos, and M. G. C. Resende. Cham: Springer International Publishing, 2018, pp. 341–367.

[47] T. L. Falch and A. C. Elster. "Machine learning based auto-tuning for enhanced OpenCL performance portability". In: *2015 IEEE international parallel and distributed processing symposium workshop*. Hyderabad, India: IEEE, May 2015, pp. 1231–1240.

[48] J. Filipovič, J. Hozzová, A. Nezarat, J. Ol'ha, and F. Petrovič. "Using hardware performance counters to speed up autotuning convergence on GPUs". In: *Journal of Parallel and Distributed Computing* 160 (Feb. 2022), pp. 16–35.

[49] J. Filipovič, F. Petrovič, and S. Benkner. "Autotuning of OpenCL kernels with global optimizations". In: *Proceedings of the 1st workshop on autotuning and aDaptivity approaches for energy efficient HPC systems*. 2017.

[50] M. Frigo and S. G. Johnson. "FFTW: An adaptive software architecture for the FFT". In: *Acoustics, speech and signal processing, 1998. Proceedings of the 1998 IEEE international conference on*. Vol. 3. IEEE, 1998, pp. 1381–1384.

[51] F. N. Fritsch and J. Butland. "A method for constructing local monotone piecewise cubic interpolants". In: *SIAM Journal on Scientific and Statistical Computing* 5.2 (1984), pp. 300–304.

[52] G. Fursin. "Collective knowledge: organizing research projects as a database of reusable components and portable workflows with common interfaces". In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 379.2197 (May 17, 2021), p. 20200211.

[53] G. Fursin, R. Miceli, A. Lokhmotov, M. Gerndt, M. Baboulin, A. D. Malony, et al. "Collective mind: Towards practical and collaborative auto-tuning". In: *Scientific Programming* 22.4 (Oct. 1, 2014), pp. 309–329.

[54] E. C. Garrido-Merchán and D. Hernández-Lobato. "Dealing with categorical and integer-valued variables in Bayesian Optimization with Gaussian processes". In: *Neurocomputing* 380 (Mar. 2020), pp. 20–35.

[55] M. Gendreau and J.-Y. Potvin. "Tabu search". In: *Search methodologies: introductory tutorials in optimization and decision support techniques*. Springer, 2013, pp. 243–263.

[56] A. Georges, D. Buytaert, and L. Eeckhout. "Statistically rigorous java performance evaluation". In: *Proceedings of the 22nd annual ACM SIGPLAN conference on object-oriented programming systems, languages and applications*. OOPSLA '07. New York, NY, USA: Association for Computing Machinery, 2007, pp. 57–76.

[57] B. W. Goldman and W. F. Punch. "Parameter-less population pyramid". In: *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*. GECCO '14. New York, NY, USA: Association for Computing Machinery, July 12, 2014, pp. 785–792.

[58] R. Goncalves, T. van Tilburg, K. Kyzirakos, F. Alvanaki, P. Koutsourakis, B. van Werkhoven, et al. "A spatial column-store to triangulate the netherlands on the fly." In: *Proceedings of the 24th ACM SIGSPATIAL international conference on advances in geographic information systems*. Gis '16. Burlingame, California and New York, NY, USA: ACM, 2016, 80:1–80:4.

[59] N. Gould and J. Scott. "A Note on Performance Profiles for Benchmarking Software". In: *ACM Transactions on Mathematical Software* 43.2 (Aug. 16, 2016), 15:1–15:5.

[60] J. Guerreiro, A. Ilic, N. Roma, and P. Tomás. "Multi-kernel auto-tuning on GPUs: Performance and energy-aware optimization". In: *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. IEEE, 2015, pp. 438–445.

[61] N. Hansen, A. Auger, R. Ros, O. Mersmann, T. Tušar, and D. Brockhoff. "COCO: a platform for comparing continuous optimizers in a black-box setting". In: *Optimization Methods and Software* 36.1 (Jan. 2, 2021), pp. 114–144.

[62] A. Hartono, B. Norris, and P. Sadayappan. "Annotation-based empirical performance tuning using Orio". In: *2009 IEEE international symposium on parallel distributed processing*. Rome, Italy, IEEE, 2009, pp. 1–11.

[63] A. B. Hayes, L. Li, D. Chavarría-Miranda, S. L. Song, and E. Z. Zhang. "Orion: A framework for gpu occupancy tuning". In: *Proceedings of the 17th International Middleware Conference*. 2016, pp. 1–13.

[64] Q. He and L. Wang. "An effective co-evolutionary particle swarm optimization for constrained engineering design problems". In: *Engineering applications of artificial intelligence* 20.1 (2007), pp. 89–99.

[65] T. Head, M. Kumar, H. Nahrstaedt, G. Louppe, and I. Shcherbatyi. *SciKit-Optimize*. Sept. 2020.

[66] S. Heldens, P. Hijma, B. V. Werkhoven, J. Maassen, A. S. Belloum, and R. V. Van Nieuwpoort. "The landscape of exascale research: a data-driven literature analysis". In: *ACM Computing Surveys (CSUR)* 53.2 (2020), pp. 1–43.

[67] S. Heldens and B. van Werkhoven. *Kernel Launcher: C++ Library for Optimal-Performance Portable CUDA Applications*. Mar. 22, 2023.

[68] E. O. Hellsten, A. Souza, J. Lenfers, R. Lacouture, O. Hsu, A. Ejjeh, et al. "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*. Asplos '23. New York, NY, USA: Association for Computing Machinery, Feb. 2024, pp. 19–42.

[69] H. Heydarian, F. Schueder, M. T. Strauss, B. van Werkhoven, M. Fazel, K. A. Lidke, et al. "Template-free 2D particle fusion in localization microscopy". In: *Nature methods* 15.10 (Oct. 2018), pp. 781–784.

[70] P. Hijma, S. Heldens, A. Sclocco, B. Van Werkhoven, and H. E. Bal. "Optimization Techniques for GPU Programming". In: *ACM Computing Surveys* 55.11 (Nov. 30, 2023), pp. 1–81.

[71] T. Hoefler and R. Belli. "Scientific benchmarking of parallel computing systems: twelve ways to tell the masses when reporting performance results". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '15. New York, NY, USA: Association for Computing Machinery, Nov. 15, 2015, pp. 1–12.

[72] M. Hoffman, E. Brochu, and N. de Freitas. "Portfolio allocation for Bayesian optimization". In: *Proceedings of the twenty-seventh conference on uncertainty in artificial intelligence*. UAI'11. Barcelona, Spain: AUAI Press, 2011, pp. 327–336.

[73] J. N. Hooker. "Testing heuristics: We have it all wrong". In: *Journal of Heuristics* 1.1 (Sept. 1, 1995), pp. 33–42.

[74] V. Horký, P. Libič, A. Steinhauser, and P. Tůma. "DOs and DON'Ts of conducting performance measurements in Java". In: *Proceedings of the 6th ACM/SPEC international conference on performance engineering*. ICPE '15. New York, NY, USA: Association for Computing Machinery, 2015, pp. 337–340.

[75] J. Hozzová, J. O. Tørring, B. van Werkhoven, D. Strelák, and R. Vuduc. "FAIR sharing of data in autotuning research (vision paper)". In: *Companion of the 15th ACM/SPEC international conference on performance engineering*. ICPE '24 companion. New York, NY, USA: Association for Computing Machinery, 2024, pp. 21–27.

[76] D. Jasrasaria and E. O. Pyzer-Knapp. "Dynamic control of explore/exploit trade-off in bayesian optimization". In: *Intelligent computing*. Ed. by K. Arai, S. Kapoor, and R. Bhatia. Cham: Springer International Publishing, 2019, pp. 1–15.

[77] T. T. Joy, S. Rana, S. Gupta, and S. Venkatesh. "Hyperparameter tuning for big data using Bayesian optimisation". In: *2016 23rd international conference on pattern recognition (ICPR)*. IEEE, 2016, pp. 2574–2579.

[78] R. Karlsson, L. Bliek, S. Verwer, and M. d. Weerdt. *Continuous surrogate-based optimization algorithms are well-suited for expensive discrete problems*. 2020.

[79] T. Kisuki, P. M. Knijnenburg, and M. F. O'Boyle. "Combined selection of tile sizes and unroll factors using iterative compilation". In: *The Journal of Supercomputing* 24.1 (2003), pp. 43–67.

[80] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. E. Emam, et al. "Preliminary guidelines for empirical research in software engineering". In: *IEEE Transactions on Software Engineering* 28.8 (Aug. 1, 2002), pp. 721–734.

[81] K. Koski, P. Manninen, and T. Nyrönen. "Fifty years of high-performance computing in finland". In: *Impact of scientific computing on science and society*. Ed. by P. Neittaanmäki and M.-L. Rantalainen. Cham: Springer International Publishing, 2023, pp. 347–355.

[82] M. Kraus. *Bayesian Optimization for quicker hyperparameter tuning*. Mar. 2019.

[83] H. J. Kushner. "A New Method of Locating the Maximum Point of an Arbitrary Multipeak Curve in the Presence of Noise". In: *Journal of Basic Engineering* 86.1 (Mar. 1964), pp. 97–106.

[84] J. Lawson, M. Goli, D. McBain, D. Soutar, and L. Sugy. "Cross-platform performance portability using highly parametrized SYCL kernels". In: *arXiv preprint arXiv:1904.05347* (2019).

[85] Q. Le and A. Smola. "Fastfood: Approximating Kernel Expansions in Loglinear Time". In: *30th international conference on machine learning (ICML)*. International Conference on Machine Learning (ICML). Google Knowledge, 2013.

[86] J.-Y. Le Boudec. *Performance Evaluation of Computer and Communication Systems*. Lausanne: EPFL Press, 2010.

[87] Y. LeCun, Y. Bengio, and G. Hinton. "Deep learning". In: *nature* 521.7553 (2015), pp. 436–444.

[88] L. Li, K. Jamieson, A. Rostamizadeh, E. Gonina, J. Ben-Tzur, M. Hardt, et al. "A system for massively parallel hyperparameter tuning". In: *Proceedings of machine learning and systems* 2 (2020), pp. 230–246.

[89] X. Li and G. Agrawal. "Shrinking Sample Search Algorithm for Automatic Tuning of GPU Kernels". In: *2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. 2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC). Bengaluru, India: IEEE, Dec. 2021, pp. 262–271.

[90] Y. Li, J. Dongarra, and S. Tomov. "A note on auto-tuning GEMM for GPUs". In: *Computational science–ICCS 2009: 9th international conference baton rouge, la, USA, may 25-27, 2009 proceedings, part I 9*. Springer, 2009, pp. 884–892.

[91] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, et al. "Competition-level code generation with AlphaCode". In: *Science* 378.6624 (Dec. 2022), pp. 1092–1097.

[92] J. Liang, X. Ban, K. Yu, B. Qu, K. Qiao, C. Yue, et al. "A survey on evolutionary constrained multiobjective optimization". In: *IEEE Transactions on Evolutionary Computation* 27.2 (2022), pp. 201–221.

[93] R. Lim, B. Norris, and A. Malony. "Autotuning GPU kernels via static and predictive analysis". In: *2017 46th international conference on parallel processing (ICPP)*. Los Alamitos, CA, USA: IEEE Computer Society, 2017, pp. 523–532.

[94] M. Liu, Yuan, Xialiang, and Tong Fei. "Evolution of heuristics: Towards efficient automatic algorithm design using large language model". In: *International conference on machine learning (ICML)*. 2024.

[95] Y. Liu, W. M. Sid-Lakhdar, O. Marques, X. Zhu, C. Meng, J. W. Demmel, et al. "GPTune: Multitask Learning for Autotuning Exascale Applications". In: *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '21. New York, NY, USA: Association for Computing Machinery, 2021, pp. 234–246.

[96] D. Lizotte. *Practical Bayesian Optimization*. University of Alberta, 2008.

[97] G. LLC. *ortools: Google OR-Tools python libraries and modules*. Version 9.7.2996. Sept. 8, 2015.

[98] P. Luong, S. Gupta, D. Nguyen, S. Rana, and S. Venkatesh. "Bayesian Optimization with Discrete Variables". In: *Australasian Conference on Artificial Intelligence*. 2019.

[99] M. Lurati, S. Heldens, A. Sclocco, and B. Van Werkhoven. "Bringing Auto-Tuning to HIP: Analysis of Tuning Impact and Difficulty on AMD and Nvidia GPUs". In: *Euro-Par 2024: Parallel Processing*. Ed. by J. Carretero, S. Shende, J. Garcia-Blas, I. Brandic, K. Olcoz, and M. Schreiber. Vol. 14801. Cham: Springer Nature Switzerland, 2024, pp. 91–106.

[100] R. Luss, S. Rosset, and M. Shahar. "Efficient Regularized Isotonic Regression with Application to Gene-Gene Interaction Search". In: *The Annals of Applied Statistics* 6.1 (2012), pp. 253–283.

[101] J. A. Martinez, E. M. Garzón, A. Plaza, and I. Garcia. "Automatic tuning of iterative computation on heterogeneous multiprocessors with ADITHE". In: *The Journal of Supercomputing* 58 (2011), pp. 151–159.

[102] P. Mattson, V. J. Reddi, C. Cheng, C. Coleman, G. Diamos, D. Kanter, et al. "MLPerf: An industry standard benchmark suite for machine learning performance". In: *IEEE Micro* 40.2 (Mar. 2020), pp. 8–16.

[103] O. Mersmann, M. Preuss, and H. Trautmann. "Benchmarking Evolutionary Algorithms: Towards Exploratory Landscape Analysis". In: *Parallel Problem Solving from Nature, PPSN XI*. Ed. by R. Schaefer, C. Cotta, J. Kołodziej, and G. Rudolph. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2010, pp. 73–82.

[104] Z. Michalewicz and G. Nazhiyath. "Genocop III: A co-evolutionary algorithm for numerical optimization problems with nonlinear constraints". In: *Proceedings of 1995 IEEE international conference on evolutionary computation*. Vol. 2. IEEE, 1995, pp. 647–651.

[105] G. G. Mitchell, D. O'Donoghue, D. Barnes, and M. McCarville. "GeneRepair - a repair operator for genetic algorithms". In: *Genetic and evolutionary computation - GECCO 2003*. 2003.

[106] T. Miyazaki, I. Sato, and N. Shimizu. "Bayesian Optimization of HPC Systems for Energy Efficiency". In: *High Performance Computing*. Ed. by R. Yokota, M. Weiland, D. Keyes, and C. Trinitis. Cham: Springer International Publishing, 2018, pp. 44–62.

[107] M. Mlakar, T. Tušar, and B. Filipic. "Comparing Random Forest and Gaussian Process Modeling in the Gp-demo Algorithm". In: *Information Security Education Journal (ISEJ)* 6.1 (June 2019), p. 9.

[108] J. Mockus. "On Bayesian Methods for Seeking the Extremum". In: *Optimization Techniques* (1974), pp. 400–404.

[109] J. J. Moré and S. M. Wild. "Benchmarking Derivative-Free Optimization Algorithms". In: *SIAM Journal on Optimization* 20.1 (Jan. 2009), pp. 172–191.

[110] D. Nan, X. Wei, J. Xu, X. Haoyu, and S. Zhenya. "CESMTuner: An auto-tuning framework for the community earth system model". In: *2014 IEEE intl conf on high performance computing and communications, 2014 IEEE 6th intl symp on cyberspace safety and security, 2014 IEEE 11th intl conf on embedded software and syst (HPCC, CSS, ICESS)*. IEEE, 2014, pp. 282–289.

[111] L. Nardi, A. Souza, D. Koeplinger, and K. Olukotun. "HyperMapper: a Practical Design Space Exploration Framework". In: *2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. 2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS). IEEE, Oct. 2019, pp. 425–426.

[112] G. Niemeyer, S. Celles, and F.-J. Willemsen. *Python-Constraint*. Version 2.4.0. July 7, 2005.

[113] F. Nogueira. *Bayesian Optimization: Open source constrained global optimization tool for Python*. 2014.

[114] C. Nugteren and V. Codreanu. "CLTune: a generic auto-tuner for OpenCL kernels". In: *2015 IEEE 9th international symposium on embedded multicore/many-core systems-on-chip (MCSoC)*. 2015, pp. 195–202.

[115] C. Nugteren. "CLBlast". In: *Proceedings of the International Workshop on OpenCL*. ACM, May 2018.

[116] P. de Oliveira Castro, E. Petit, J. C. Beyler, and W. Jalby. "ASK: Adaptive Sampling Kit for Performance Characterization". In: *Euro-Par 2012 Parallel Processing*. Ed. by C. Kaklamanis, T. Papatheodorou, and P. G. Spirakis. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 89–101.

[117] A. V. Papadopoulos, L. Versluis, A. Bauer, N. Herbst, J. v. Kistowski, A. Ali-Eldin, et al. "Methodological Principles for Reproducible Performance Evaluation in Cloud Computing". In: *IEEE Transactions on Software Engineering* 47.8 (Aug. 1, 2021), pp. 1528–1543.

[118] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, et al. "Scikit-learn: Machine learning in Python". In: *Journal of machine learning research* 12 (Oct 2011), pp. 2825–2830.

[119] M. Pelikan. "Analysis of estimation of distribution algorithms and genetic algorithms on NK landscapes". In: *Proceedings of the 10th annual conference on Genetic and evolutionary computation*. GECCO '08. New York, NY, USA: Association for Computing Machinery, July 12, 2008, pp. 1033–1040.

[120] F. Petrovič and J. Filipovič. "Kernel tuning toolkit". In: *SoftwareX* 22 (2023), p. 101385.

[121] F. Petrovič, D. Střelák, J. Hozzová, J. Ol'ha, R. Trembecký, S. Benkner, et al. "A benchmark set of highly-efficient CUDA and OpenCL kernels and its dynamic autotuning with Kernel Tuning Toolkit". In: *Future Generation Computer Systems* 108 (July 1, 2020), pp. 161–177.

[122] R. Pincus, E. J. Mlawer, and J. S. Delamere. "Balancing Accuracy, Efficiency, and Flexibility in Radiation Calculations for Dynamical Models". In: *Journal of Advances in Modeling Earth Systems* 11.10 (Oct. 2019), pp. 3074–3089.

[123] D. C. Price, M. A. Clark, B. R. Barsdell, R. Babich, and L. J. Greenhill. "Optimizing performance-per-watt on GPUs in high performance computing". In: *Computer Science-Research and Development* 31.4 (2016), pp. 185–193.

[124] W. A. Pruett and R. L. Hester. "The Creation of Surrogate Models for Fast Estimation of Complex Model Outcomes". In: *PLOS ONE* 11.6 (2016), pp. 1–11.

[125] A. Rasch, M. Haidl, and S. Gorlatch. "ATF: A Generic Auto-Tuning Framework". In: *Proceedings of the 19th International Conference on High Performance Computing and Communications*. 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). Dec. 2017, pp. 64–71.

[126] A. Rasch, R. Schulze, M. Steuwer, and S. Gorlatch. "Efficient auto-tuning of parallel programs with interdependent tuning parameters via auto-tuning framework (ATF)". In: *ACM Trans. Archit. Code Optim.* 18.1 (Jan. 2021).

[127] C. Rasmussen and C. Williams. "Chapter 4: Covariance Functions". In: *Gaussian Processes for Machine Learning*. The MIT Press, 2006, pp. 79–104.

[128] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, and C.-J. Wu. "MLPerf inference benchmark". In: *2020 ACM/IEEE 47th annual international symposium on computer architecture (ISCA)*. May 2020, pp. 446–459.

[129] B. Romera-Paredes, M. Barekatain, A. Novikov, M. Balog, M. P. Kumar, E. Dupont, et al. "Mathematical discoveries from program search with large language models". In: *Nature* 625.7995 (2024), pp. 468–475.

[130] B. Ru, A. Alvi, V. Nguyen, M. A. Osborne, and S. Roberts. "Bayesian Optimisation over Multiple Continuous and Categorical Inputs". In: *Proceedings of the 37th International Conference on Machine Learning*. Ed. by H. D. III and A. Singh. Vol. 119. Proceedings of Machine Learning Research. PMLR, July 13, 2020, pp. 8276–8285.

[131] Y. Rubanova, D. Dohan, K. Swersky, and K. Murphy. "Amortized Bayesian Optimization over Discrete Spaces". In: *Proceedings of Machine Learning Research / Proceedings of the 36th Conference on Uncertainty in Artificial Intelligence (UAI)* 124 (2020), pp. 769–778.

[132] T. P. Runarsson and X. Yao. "Stochastic ranking for constrained evolutionary optimization". In: *IEEE Transactions on Evolutionary Computation* 4.3 (2000), pp. 284–294.

[133] S. Ryoo, C. Rodrigues, S. Stone, S. Baghsorkhi, S. Ueng, J. Stratton, et al. "Program optimization space pruning for a multithreaded gpu". In: *Code generation and optimization. International symposium on*. ACM, 2008, pp. 195–204.

[134] R. Schoonhoven, B. van Werkhoven, and K. J. Batenburg. "Benchmarking optimization algorithms for auto-tuning GPU kernels". In: *IEEE Transactions on Evolutionary Computation* (2022), pp. 1–1.

[135] R. Schulze, S. Gorlatch, and A. Rasch. "pyATF: Constraint-based auto-tuning in python". In: *Proceedings of the 34th ACM SIGPLAN international conference on compiler construction*. Cc '25. Las Vegas, NV, USA and New York, NY, USA: Association for Computing Machinery, 2025, pp. 35–47.

[136] A. Sclocco, H. E. Bal, J. Hessels, J. v. Leeuwen, and R. V. v. Nieuwpoort. "Auto-Tuning Dedispersion for Many-Core Accelerators". In: *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. 2014 IEEE International Parallel & Distributed Processing Symposium (IPDPS). Phoenix, AZ, USA: IEEE, May 2014, pp. 952–961.

[137] A. Sclocco, S. Heldens, and B. van Werkhoven. "AMBER: a real-time pipeline for the detection of single pulse astronomical transients". In: *SoftwareX* 12 (2020), p. 100549.

[138] C. Seger. "An investigation of categorical variable encoding techniques in machine learning: binary versus one-hot and feature hashing". 2018.

[139] K. Seymour, H. You, and J. Dongarra. "A comparison of search heuristics for empirical code optimization". In: *Cluster computing, 2008 IEEE international conference on*. IEEE, 2008, pp. 421–429.

[140] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, and N. d. Freitas. "Taking the Human Out of the Loop: A Review of Bayesian Optimization". In: *Proceedings of the IEEE* 104.1 (2016), pp. 148–175.

[141] J. Snoek, H. Larochelle, and R. P. Adams. *Practical Bayesian Optimization of Machine Learning Algorithms*. 2012.

[142] B. Solnik, D. Golovin, G. Kochanski, J. E. Karro, S. Moitra, and D. Sculley. "Bayesian Optimization for a Better Dessert". In: *Proceedings of the 2017 NIPS Workshop on Bayesian Optimization*. Long Beach, USA, Dec. 9, 2017.

[143] K. Spafford, J. Meredith, and J. Vetter. "Maestro: Data orchestration and tuning for OpenCL devices". In: *Euro-par 2010 - parallel processing*. Ed. by P. D'Ambra, M. Guarracino, and D. Talia. Lecture notes in computer science. Springer Berlin Heidelberg, 2010, pp. 275–286.

[144] M. Stachowski, A. Fiebig, and T. Rauber. "Autotuning based on frequency scaling toward energy efficiency of blockchain algorithms on graphics processing units". In: *The Journal of Supercomputing* 77.1 (Jan. 1, 2021), pp. 263–291.

[145] M. L. Stein. "Matern class". In: *Interpolation of Spatial Data*. Springer New York, 1999, pp. 31–33.

[146] N. van Stein and T. Bäck. *LLaMEA: a large language model evolutionary algorithm for automatically generating metaheuristics*. 2025.

[147] N. van Stein, A. V. Kononova, H. Yin, and T. Bäck. "BLADE: Benchmark suite for LLM-driven automated design and evolution of iterative optimisation heuristics". In: *GECCO '25 companion*. Proceedings of the genetic and evolutionary computation conference companion. Association for Computing Machinery, 2025.

[148] N. van Stein, D. Vermetten, and T. Bäck. "In-the-loop hyper-parameter optimization for LLM-based automated design of heuristics". In: *ACM Trans. Evol. Learn. Optim.* (Apr. 2025).

[149] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, et al. "Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing". In: *Center for Reliable and High-Performance Computing* 127 (Mar. 2, 2012), p. 27.

[150] P. D. Surry, N. J. Radcliffe, and I. D. Boyd. "A multi-objective approach to constrained optimisation of gas supply networks: The COMOGA method". In: *AISB workshop on evolutionary computing*. Springer, 1995, pp. 166–180.

[151] L. P. Swiler, P. D. Hough, P. Qian, X. Xu, C. Storlie, and H. Lee. "Surrogate Models for Mixed Discrete-Continuous Variables". In: *Constraint Programming and Decision Making* (2014), pp. 181–202.

[152] C. Tapus, I.-H. Chung, and J. Hollingsworth. "Active harmony: Towards automated performance tuning". In: *Proceedings from the conference on high performance networking and computing*. Proceedings from the Conference on High Performance Networking and Computing. Dec. 2002, pp. 44–44.

[153] P. Team. *PySMT: A solver-agnostic library for SMT Formulae manipulation and solving*. Version 0.9.5. May 28, 2022.

[154] TechPowerUp. *NVIDIA A100 SXM4 40 GB*. May 2020.

[155] TechPowerUp. *NVIDIA GeForce GTX TITAN X specifications*. Mar. 2015.

[156] TechPowerUp. *NVIDIA GeForce RTX 2070 SUPER specifications*. July 2019.

[157] B. Tessema and G. G. Yen. "A self adaptive penalty function based algorithm for constrained optimization". In: *2006 IEEE international conference on evolutionary computation*. IEEE, 2006, pp. 246–253.

[158] J. O. Tørring and A. C. Elster. "Analyzing Search Techniques for Autotuning Image-based GPU Kernels: The Impact of Sample Sizes". In: *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). May 2022, pp. 972–981.

[159] J. O. Tørring, B. van Werkhoven, F. Petrovč, F.-J. Willemsen, J. Filipovič, and A. C. Elster. "Towards a benchmarking suite for kernel tuners". In: *2023 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*. IEEE, 2023, pp. 724–733.

[160] C. C. Van Heerwaarden, B. J. Van Stratum, T. Heus, J. A. Gibbs, E. Fedorovich, and J. P. Mellado. "MicroHH 1.0: A computational fluid dynamics code for direct numerical simulation and large-eddy simulation of atmospheric boundary layer flows". In: *Geoscientific Model Development* 10.8 (2017), pp. 3145–3165.

[161] P. J. Van Laarhoven and E. H. Aarts. "Simulated annealing". In: *Simulated annealing: Theory and applications*. Springer, 1987, pp. 7–15.

[162] S. Venkatraman and G. G. Yen. "A generic framework for constrained optimization using genetic algorithms". In: *IEEE Transactions on Evolutionary Computation* 9.4 (2005), pp. 424–435.

[163] Y. Wang and B. Vinter. "Auto-tuning for large-scale image processing by dynamic analysis method on multicore platforms". In: *International Journal of Embedded Systems* 8.4 (2016), pp. 313–322.

[164] N. Weber and M. Goesele. "MATOG: Array Layout Auto-Tuning for CUDA". In: *ACM Transactions on Architecture and Code Optimization* 14.3 (Aug. 30, 2017), 28:1–28:26.

[165] B. van Werkhoven. "Kernel Tuner: A search-optimizing GPU code auto-tuner". In: *Future Generation Computer Systems* 90 (2019), pp. 347–358.

[166] B. van Werkhoven, J. Maassen, H. E. Bal, and F. J. Seinstra. "Optimizing convolution operations on GPUs using adaptive tiling". In: *Future Generation Computer Systems* 30 (Jan. 2014), pp. 14–26.

[167] B. van Werkhoven, W. J. Palenstijn, and A. Sclocco. "Lessons learned in a decade of research software engineering GPU applications". In: *Computational science – ICCS 2020*. 2020, pp. 399–412.

[168] R. C. Whaley, A. Petitet, and J. J. Dongarra. "Automated empirical optimizations of software and the ATLAS project". In: *Parallel Computing* 27.1 (2001), pp. 3–35.

[169] F.-J. Willemsen, S. Heldens, R. V. van Nieuwpoort, and B. van Werkhoven. "Constraint-aware Optimization in Auto-Tuning". In: *2026 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. New Orleans, LA, USA, 2026.

[170] F.-J. Willemsen, R. van Nieuwpoort, and B. van Werkhoven. "Bayesian Optimization for auto-tuning GPU kernels". In: *2021 International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS) at SuperComputing 2021*. 2021 International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS) at SuperComputing 2021. Nov. 2021, pp. 106–117.

[171] F.-J. Willemsen, R. V. van Nieuwpoort, and B. van Werkhoven. "Efficient construction of large search spaces for auto-tuning". In: *Proceedings of the 54th international conference on parallel processing (ICPP '25)*. San Diego, CA, USA and New York, NY, USA: Association for Computing Machinery, Sept. 2025.

[172] F.-J. Willemsen, R. V. van Nieuwpoort, and B. van Werkhoven. "Tuning the Tuner: Introducing Hyperparameter Optimization for Auto-Tuning". In: *Proceedings of the 2025 IEEE eScience conference*. 21st IEEE International Conference on eScience. Chicago, IL, USA, Sept. 2025.

[173] F.-J. Willemsen, R. Schoonhoven, J. Filipovič, J. O. Tørring, R. van Nieuwpoort, and B. van Werkhoven. "A methodology for comparing optimization algorithms for auto-tuning". In: *Future Generation Computer Systems* 159 (2024), pp. 489–504.

[174] F.-J. Willemsen, N. van Stein, and B. van Werkhoven. "Automated Algorithm Design for Auto-Tuning Optimizers". In: *Proceedings of Machine Learning and Systems*. MLSys 2026. Seattle, WA, USA, 2026.

[175] X. Wu, P. Balaprakash, M. Kruse, J. Koo, B. Videau, P. Hovland, et al. "ytopt: Autotuning Scientific Applications for Energy Efficiency at Large Scales". In: *Concurrency and Computation: Practice and Experience* (Oct. 30, 2024), e8322.

[176] X. Wu, M. Kruse, P. Balaprakash, H. Finkel, P. Hovland, V. Taylor, et al. "Autotuning PolyBench Benchmarks with LLVM Clang/Polly Loop Optimization Pragmas Using Bayesian Optimization". In: *2020 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. 2020 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS). Nov. 2020, pp. 61–70.

[177] Yadolah Dodge. "Central Limit Theorem". In: *The Concise Encyclopedia of Statistics*. New York, NY: Springer, 2008, pp. 66–68.

[178] X.-S. Yang. *Nature-inspired optimization algorithms*. Academic Press, 2020.

[179] H. Ye, J. Wang, Z. Cao, F. Berto, C. Hua, H. Kim, et al. "Reevo: Large language models as hyperheuristics with reflective evolution". In: *Advances in neural information processing systems* 37 (2024), pp. 43571–43608.

[180] F. Zehra, M. Javed, D. Khan, and M. Pasha. *Comparative analysis of C++ and python in terms of memory and time*. 2020.

Summary

Modern scientific discoveries and technologies increasingly rely on powerful computers. From predicting the climate to training artificial intelligence, researchers depend on High-Performance Computing (HPC) systems that combine traditional processors with specialized accelerators such as graphics cards (GPUs). These machines are extremely fast, but getting the most out of them is surprisingly difficult. Small choices in how a program is written, such as how data is stored, how computation is organized, or how many threads run in parallel, can make performance vary wildly. Because the number of possible choices is enormous, automatic performance tuning (auto-tuning) is needed: letting the computer automatically test different program configurations until it finds a fast and efficient one. Auto-tuning has proven highly effective, but it also introduces a new problem: how do we best design the tuning process itself? Choosing how to search, which parameters to test, and how to handle constraints can make the difference between success and failure.

This thesis focuses on improving auto-tuning for GPUs, which are widely used in scientific and industrial computing. The research advances the field in six directions:

- Developing a new optimization method based on Bayesian statistics, designed specifically for GPU tuning, which outperforms existing approaches.
- Creating a fair way to compare tuning methods, ensuring that researchers can reliably judge which approach works best and improve upon it.
- Building a fast solver for search spaces, capable of handling billions of possible configurations in less than a second, removing a major bottleneck.
- Showing how tuning the tuners themselves can lead to performance improvements of more than 200%, while reducing energy use through large-scale simulations.
- Making evolutionary algorithms more constraint-aware, so they can better navigate the complex rules that govern GPU programs.
- Exploring whether artificial intelligence can invent new optimizers, showing that machine-generated algorithms can beat human-designed ones.

Together, these contributions show that, as big improvements can come from auto-tuning programs, improving the underlying tuning systems themselves can enable the next generation of technological breakthroughs. All the methods and tools from this thesis have been released as open-source software, enabling researchers and practitioners worldwide to benefit from them.

Looking ahead, the results point to a future where auto-tuning becomes smarter, more efficient, and more adaptive. Optimizers that understand constraints, balance performance with energy use, or even design themselves with the help of AI could make HPC more accessible and sustainable. In this way, this work contributes to enabling faster progress in science and technology, powered by computers that tune themselves to the challenges of tomorrow.

Samenvatting

Moderne wetenschappelijke ontdekkingen en technologieën zijn in toenemende mate afhankelijk van krachtige computers. Van het voorspellen van het klimaat tot het trainen van kunstmatige intelligentie vertrouwen onderzoekers op High-Performance Computing (HPC)-systemen die traditionele processoren combineren met gespecialiseerde accelerators zoals grafische kaarten (GPU's). Deze machines zijn extreem snel, maar het is verrassend lastig om er het maximale uit te halen. Kleine keuzes in de manier waarop een programma wordt geschreven, zoals hoe data worden opgeslagen, hoe berekeningen worden georganiseerd, of hoeveel parallele verwerkingen er zijn, kunnen leiden tot grote verschillen in prestaties.

Omdat het aantal mogelijke keuzes enorm is, bestaat het automatisch verbeteren van prestaties (*auto-tuning*): de computer zelf automatisch verschillende configuraties van een programma laten testen totdat er een snelle en efficiënte variant wordt gevonden. Autotuning is zeer effectief gebleken, maar introduceert ook een nieuw probleem: hoe ontwerpen we dit afstemmingsproces zelf het beste? De keuze van de zoekstrategie, welke waarden getest worden en hoe om te gaan met ongeldige configuraties kan een groot verschil maken.

Dit proefschrift richt zich op het verbeteren van auto-tuning voor GPU's, die veel worden gebruikt in zowel wetenschappelijke als industriële toepassingen. Het onderzoek levert bijdragen in zes richtingen:

- Ontwikkeling van een nieuwe optimalisatiemethode gebaseerd op bayesiaanse statistiek, speciaal ontworpen voor GPU-tuning, die bestaande benaderingen overtreft.

- Het creëren van een eerlijke manier om tuningmethoden te vergelijken, zodat onderzoekers betrouwbaar kunnen beoordelen welke aanpak het beste werkt.

- Bouw van een snelle oplosser voor zoekruimtes, die miljarden mogelijke configuraties in minder dan een seconde kan verwerken en zo een belangrijke bottleneck wegneemt.

- Aantonen dat het tunen van de tuners zelf kan leiden tot prestatieverbeteringen van meer dan 200%, terwijl het energieverbruik wordt verminderd via grootschalige simulaties.
- Evolutionaire algoritmen beter beperking-bewust maken, zodat zij effectiever kunnen omgaan met complexe beperkingen die inherent zijn aan GPU-programma's.
- Onderzoeken of kunstmatige intelligentie nieuwe optimalisatiealgoritmen kan uitvinden, en laten zien dat deze automatisch ontworpen algoritmes beter kunnen presteren dan die van mensen.

Gezamenlijk laten deze bijdragen zien dat het verbeteren van de onderliggende tuningsystemen zelf de volgende generatie technologische doorbraken kan voortbrengen. Alle methoden en hulpmiddelen uit dit proefschrift zijn als open-source software beschikbaar gesteld, zodat onderzoekers en praktijkmensen wereldwijd ervan kunnen profiteren.

Vooruitkijkend wijzen de resultaten op een toekomst waarin auto-tuning slimmer, efficiënter en adaptiever wordt. Algoritmen die beperkingen van de zoekruimte begrijpen, prestaties afwegen tegen energieverbruik, of zelfs zichzelf ontwerpen met behulp van AI, kunnen rekenkracht op grote schaal toegankelijker en duurzamer maken. Op die manier draagt dit werk bij aan vooruitgang in wetenschap en technologie, aangedreven door computers die zichzelf afstemmen op de uitdagingen van morgen.

Acknowledgements

Pursuing a Ph.D. is often described as a solitary endeavour, carried out in quiet rooms and for long hours, with only the rhythm of thought and keyboard for company. Yet no journey of this length is truly travelled alone, and this work would not have been possible without the support, encouragement, and generosity of many people who walked beside me in ways both visible and unseen.

I want to begin by expressing my sincere gratitude to my daily supervisor, Ben van Werkhoven. His expertise, guidance, and steady availability have shaped this dissertation in countless ways, and his trust allowed me the freedom to explore, to question, and to grow as a researcher. I am also grateful to my advisor, Rob van Nieuwpoort, whose clear vision, thoughtful feedback, and unwavering support have anchored this undertaking from beginning to end. Working under the supervision of both Ben and Rob has been a privilege, and one that I will continue to appreciate long after these pages are bound.

This research was made possible through the CORTEX project (with thanks to Joeri van Leeuwen) and the Netherlands eScience Center, whose mission and environment offered a fertile ground for learning and collaboration, in which I was facilitated by Nicolas Renaud and Gijs van den Oord. Stijn and Alessio, thank you for the memorable trips, the inspiring conversations, and the many lessons that came simply from watching how you approached problems, projects, and the occasional shared adventure. I also thank the University of Amsterdam, particularly Paola Grosso, for the hospitality during the first half of my doctoral work, and Leiden University for continuing this support in the second half. My gratitude extends to Younghyun Cho, who welcomed me during my research visit at Santa Clara University and provided a lively space for exchanging ideas and perspectives.

I am also grateful for the many collaborators and experiences that made this journey far richer than I could have anticipated. The opportunity to work across universities, research centers, and industry partners has not only strengthened this research but also broadened my perspective on what our field can achieve together. Conferences, research visits, and tutorials allowed me to exchange ideas with people around the world who inspired me

with their curiosity and openness, many of whom have become valued collaborators and friends. The conversations in lecture halls, over whiteboards, during coffee breaks, and across time zones show that science is an extraordinary collective endeavor.

To my fellow Ph.D. students at the University of Amsterdam and Leiden University - particularly Marco, Leonardo, Daphnée, Damian, Miguel, Badia, Wouter, and Skip, as well as so many other awesome people - thank you for the discussions that sharpened my thinking, the shared struggles that made difficult days easier to bear, the sparks of inspiration that appeared when least expected, and the occasional beer. Peer support, I have learned, often proves far kinder and more sustaining than peer review.

Finally, I wish to express my deepest gratitude to Emma, my parents, my sisters, my brother, and so many amazing friends and dear family members. For twenty-eight years, they have offered guidance, warmth, and a place to return to, shaping me long before I ever set foot in academia. Their encouragement, patience, faith, and the balance they provided have been a foundation as enduring as bedrock. They were always there to celebrate the milestones, absorb the frustrations, and quietly supported the long stretches in between. Whatever clarity or resilience this thesis reflects is rooted in the love they continue to give.

Curriculum Vitae

Floris-Jan Willemsen was born on December 11, 1997, in Middelharnis, the Netherlands. He earned his Bachelor's degree in Information Technology at the Rotterdam University of Applied Sciences, where he graduated cum laude and completed the honors program in 2019. He subsequently completed a Master's degree in Computer Science from the University of Amsterdam and Vrije Universiteit Amsterdam in 2021.

In October 2021, he began his Ph.D. in Computer Science in collaboration with the Netherlands eScience Center, which he did at the University of Amsterdam and Leiden University under the supervision of Prof. Rob van Nieuwpoort and Dr. Ben van Werkhoven. Over the course of his Ph.D. studies, he completed courses on academic management, resource awareness, high-performance computing, real-time systems, blockchain-based systems, and scientific conduct, among others. In addition, he conducted a research visit at Santa Clara University in California, as well as a wide variety of presentations and tutorials at conferences and institutes.

Alongside his academic work, Floris-Jan has built a strong professional background through his work on open source software for science at the Netherlands eScience Center. His experience also includes freelance software development and internships in R&D and software engineering roles. He worked as a teaching assistant on courses on machine learning and high-performance computing, mentored students, and contributed to the academic community through organizing events, leading discussion groups, and initiating collaboration. His academic work has been recognized with awards at the International Conference on Parallel Processing and CompSys.