Tuning the tuner: the art of automatic performance optimization

Willemsen, F.Q.

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.

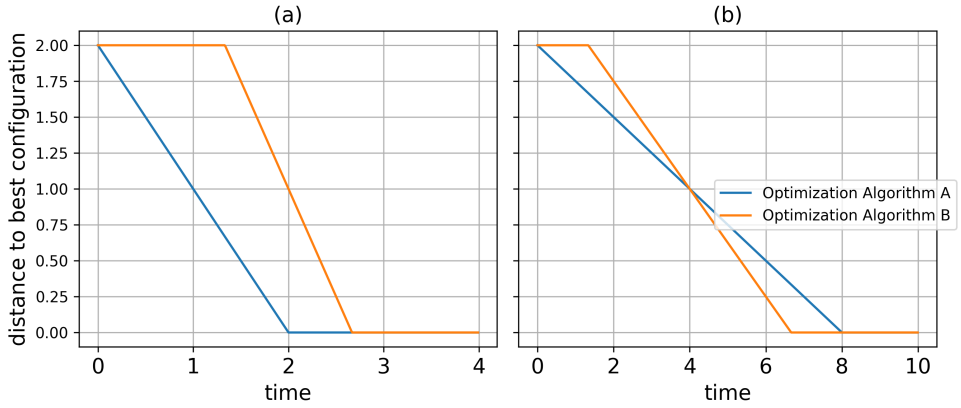


Figure 3.1: Modifying application run-time changes the outcome.

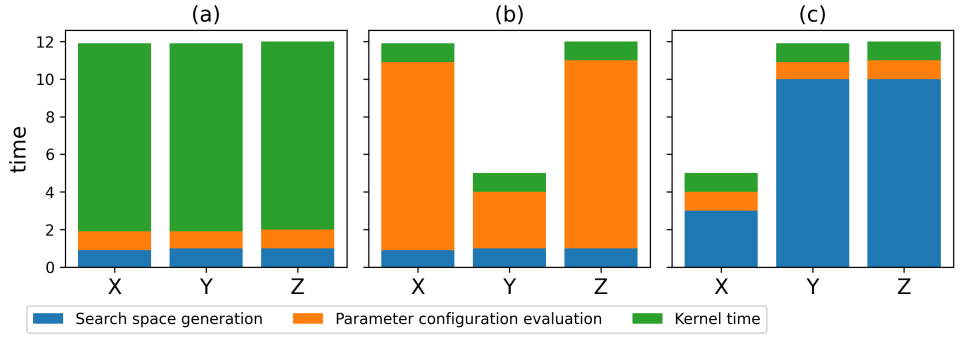


Figure 3.2: Emphasizing differences in overhead changes the outcome.

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.

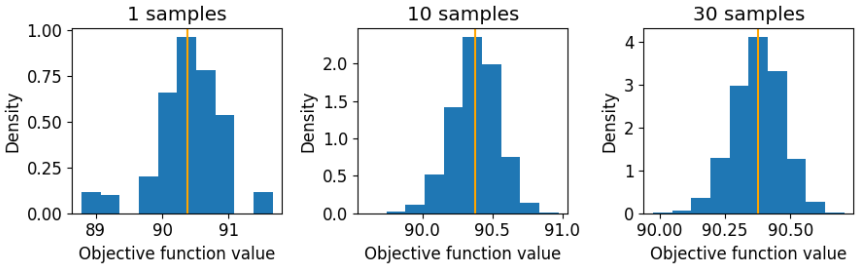


Figure 3.3: Distribution of GPU kernel execution times with three sample sizes in a density histogram, 10000 repeats, orange line is the population mean.

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

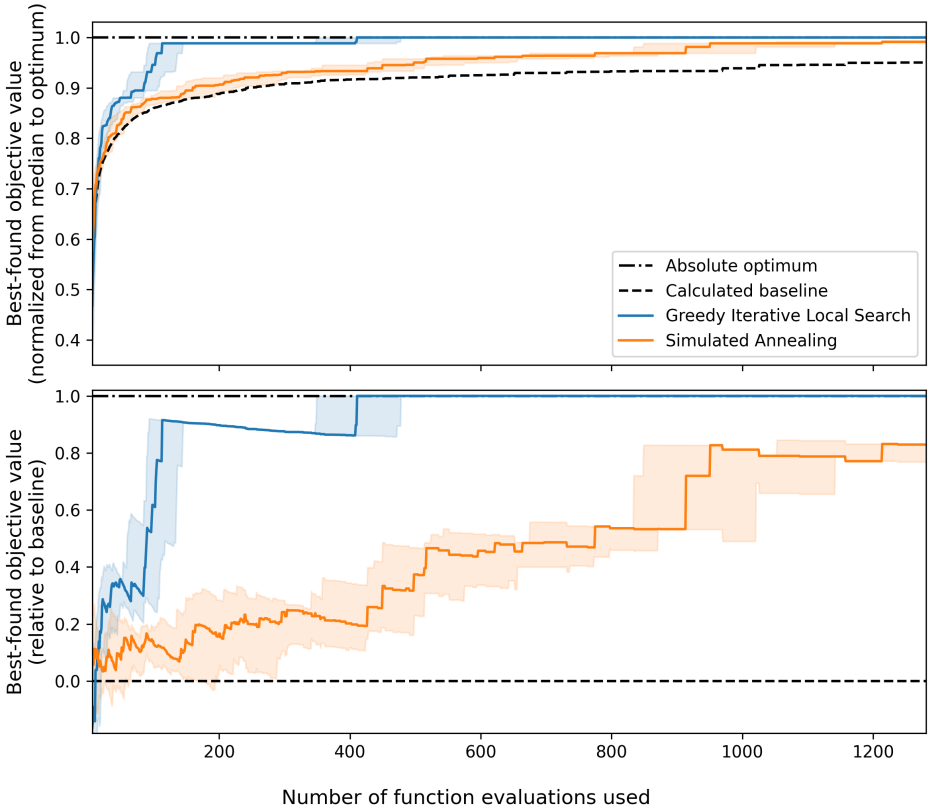


Figure 3.4: Example visualization of optimization algorithm performance over function evaluations on a single search space.

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

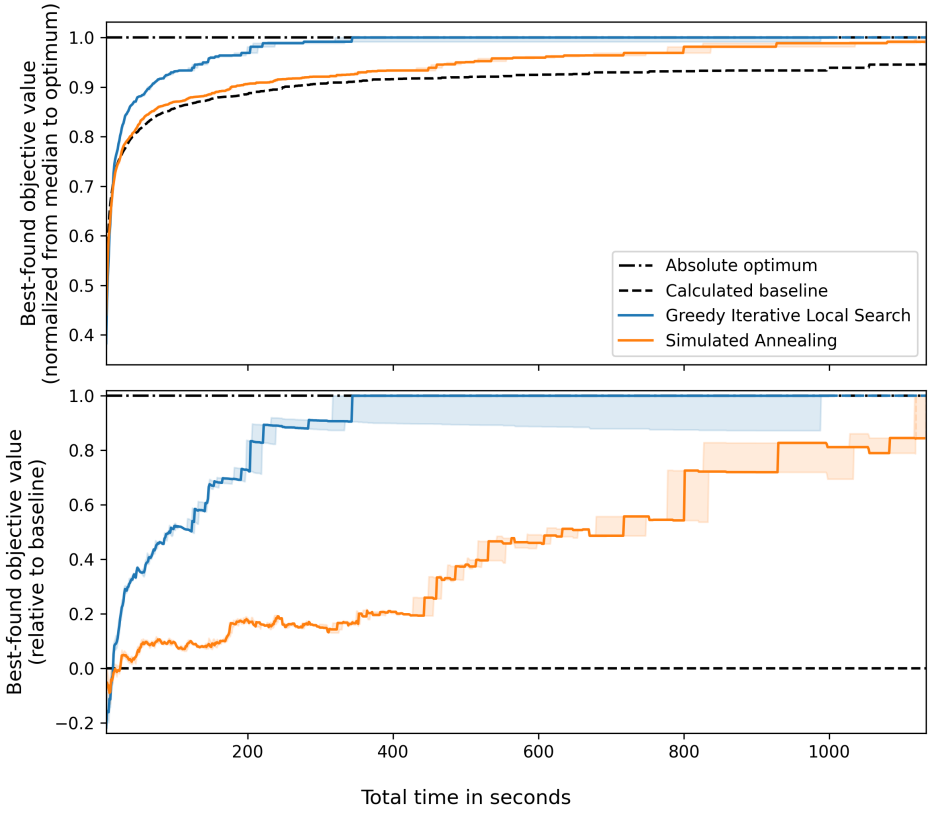


Figure 3.5: Example visualization of optimization algorithm performance over time on a single search space.

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

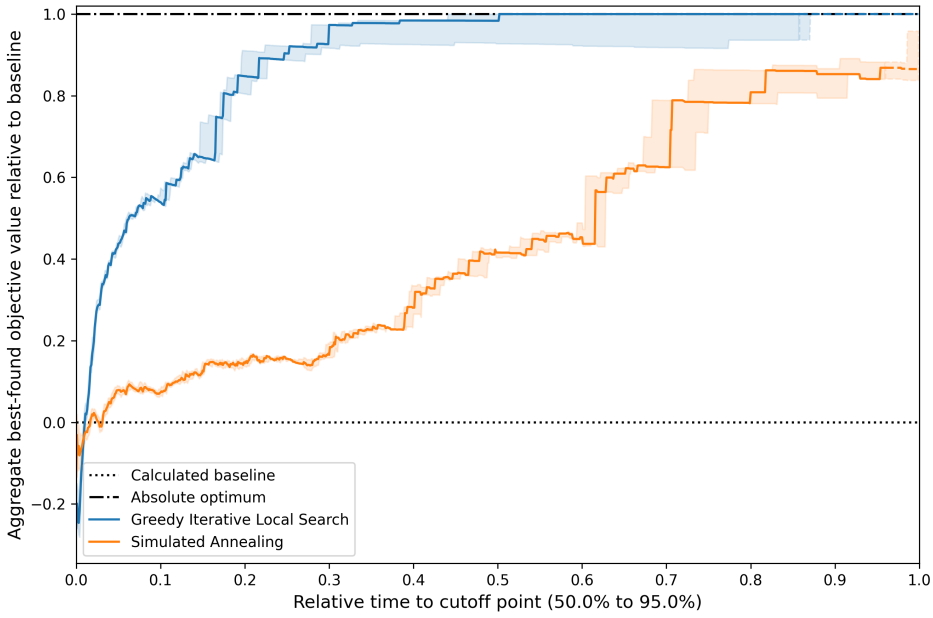
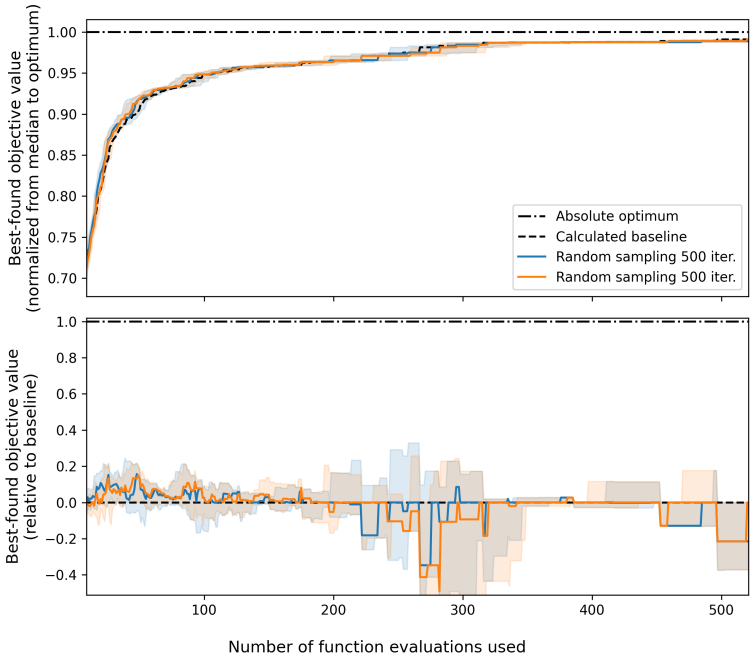


Figure 3.6: Example visualization of aggregate optimization algorithm performance across search spaces.

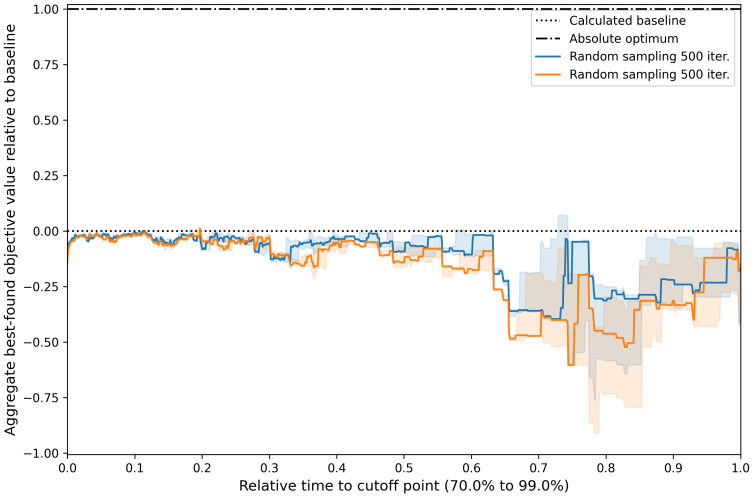
range of time must be the same for each search space. As the tuning budget of Section 3.3.3 is already relative to each search space, the time range can be made comparable across search spaces by normalizing from 0 at the start to 1 at the end of the tuning budget. If the start of this range is not of 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.

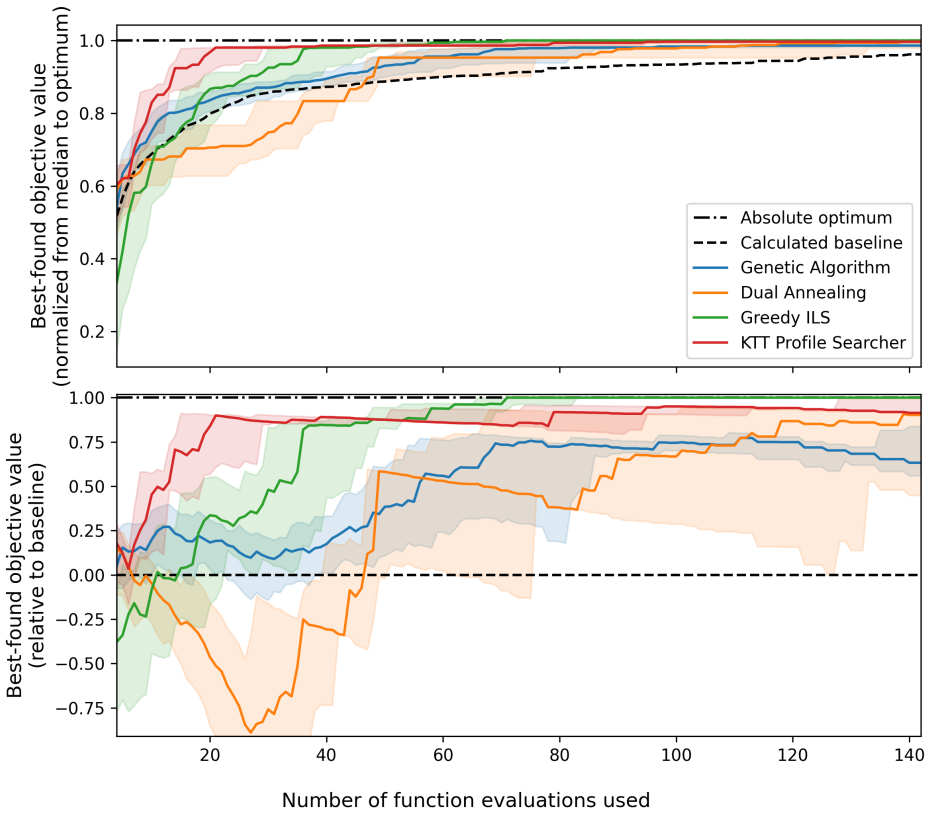


Figure 3.8: Performance over function evaluations for the *pnpoly* kernel on the RTX 2080 Ti.

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

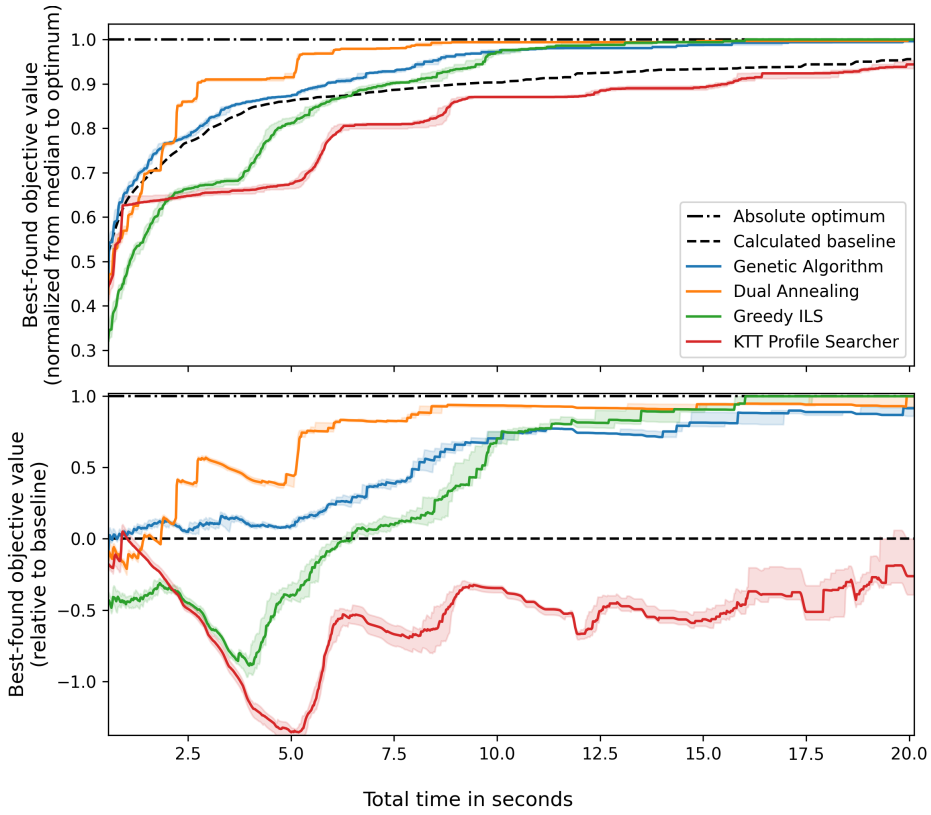


Figure 3.9: Performance over time for the *pnpoly* kernel on the RTX 2080 Ti.

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

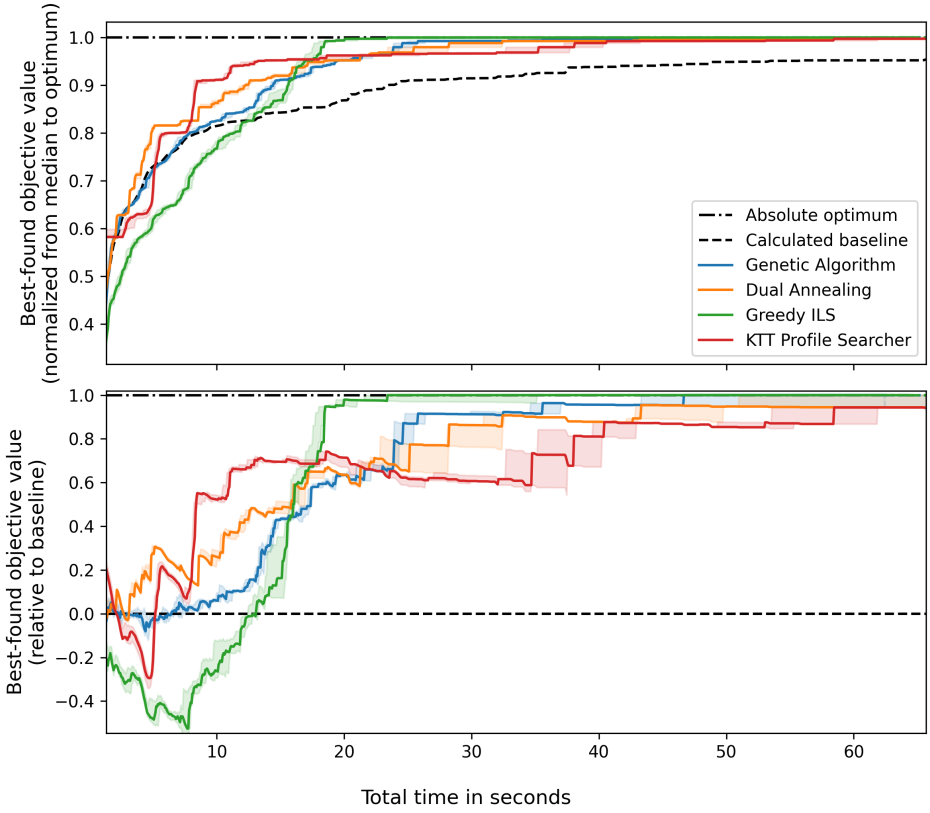


Figure 3.10: Performance on *pnpoly* on the RTX 3090.

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,

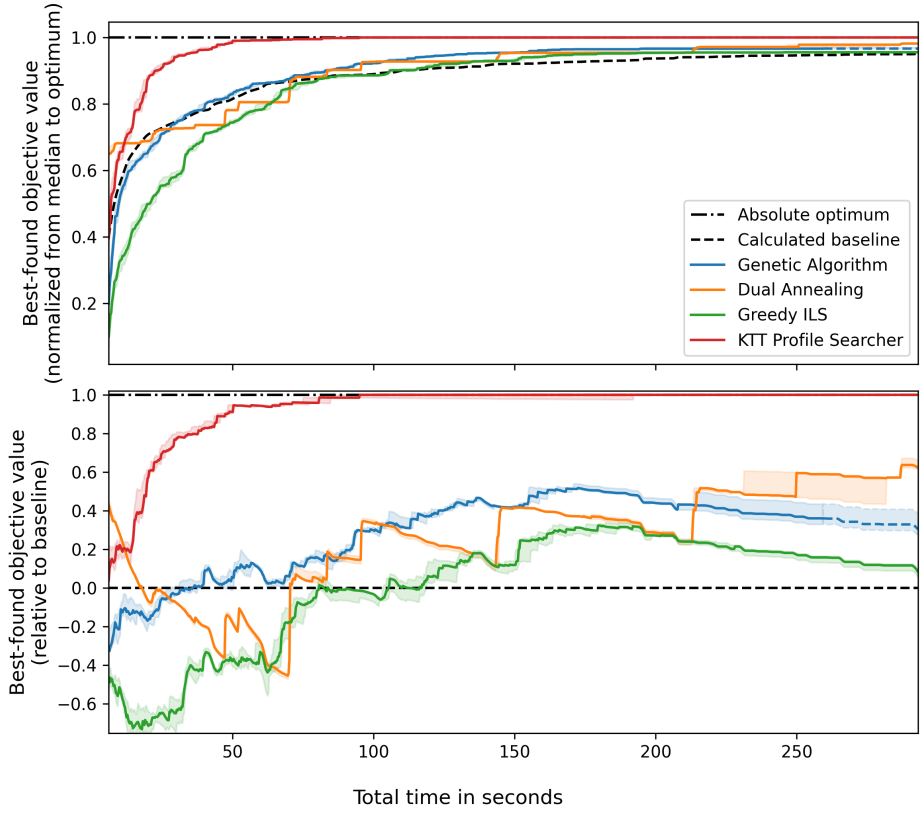


Figure 3.11: Performance on *convolution* on the RTX 3090.

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.

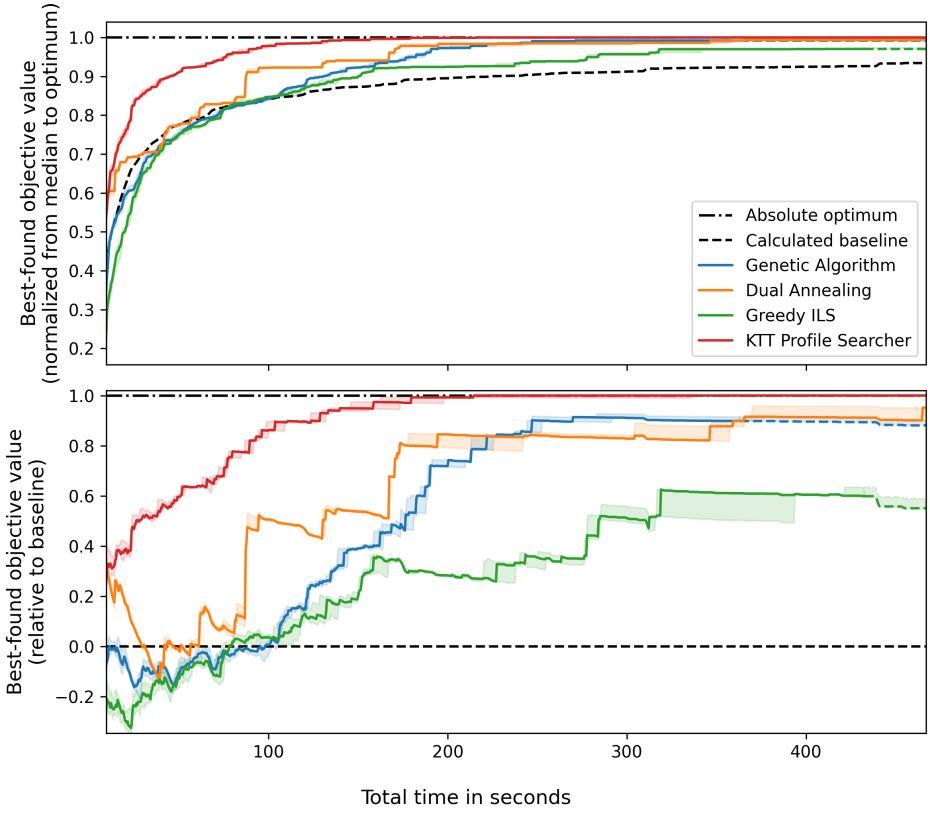


Figure 3.12: Performance on *convolution* on the RTX 2080 Ti.

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.

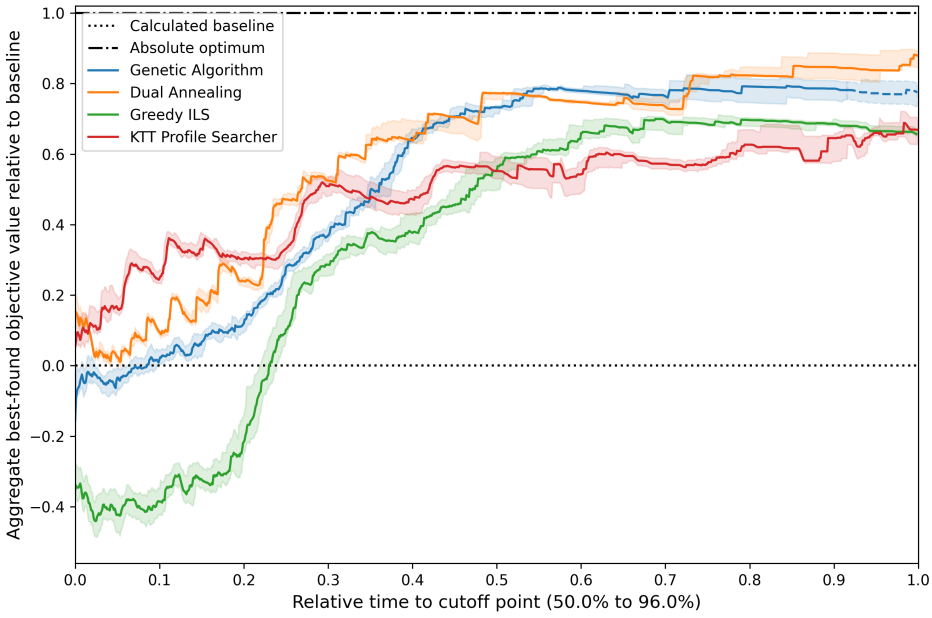


Figure 3.13: Visualization of optimization algorithm performance aggregated across both kernels and both GPUs.

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