# Tuning the tuner: the art of automatic performance optimization

Willemsen, F.Q.

# 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.

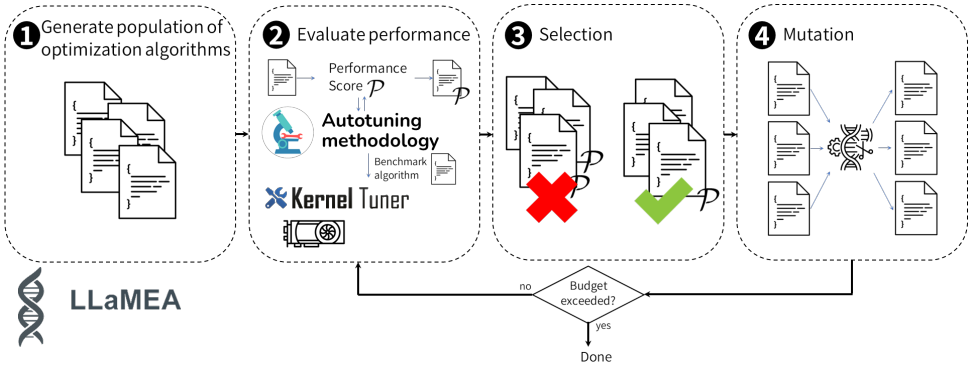



**Figure 7.1:** Overview of our designed system, showing how LLaMEA generates optimization algorithms, which are evaluated using the autotuning methodology, which tests the performance of the optimization algorithm in Kernel Tuner.

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

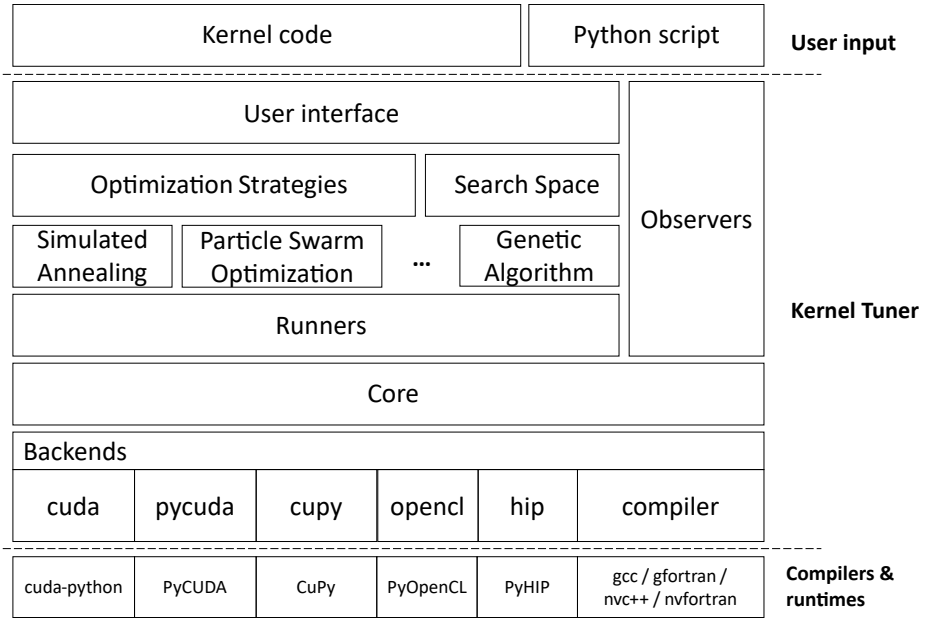



**Figure 7.2:** Overview of the software architecture of Kernel Tuner.

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 $\alpha$ and $\beta$ 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

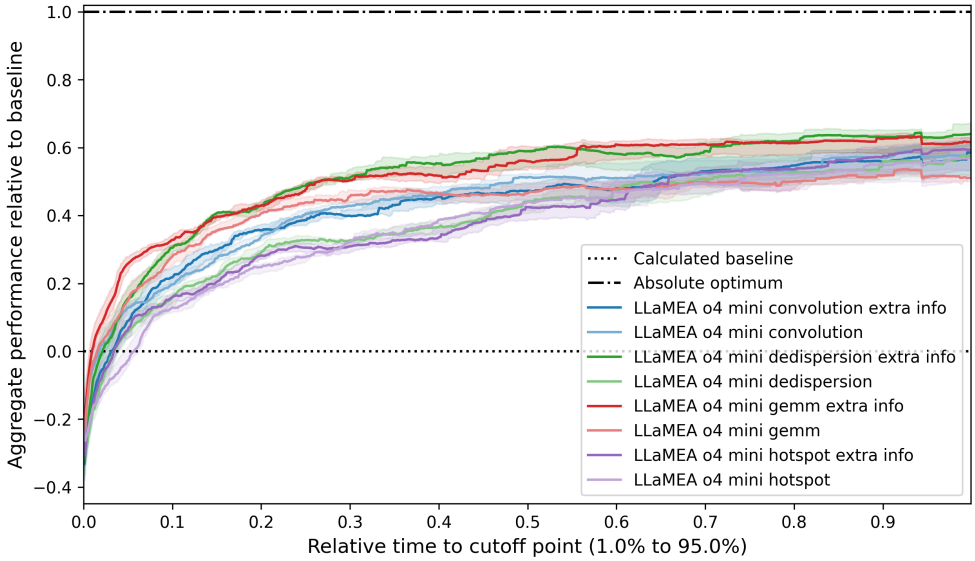



**Figure 7.5:** The aggregate performance over time over all search spaces of the application-specific optimization algorithms, each with and without additional search space knowledge. Mean of 100 runs each, the shaded area marks the 95% confidence interval.

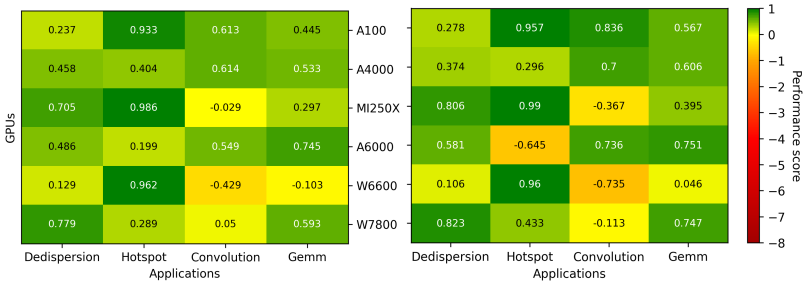
time relative to a statistically estimated *random search baseline*. This aggregate metric captures both search efficiency and solution quality across different search spaces. A score of 0.0 indicates parity with the baseline, while a score of 1.0 corresponds to consistently finding the global optimum 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

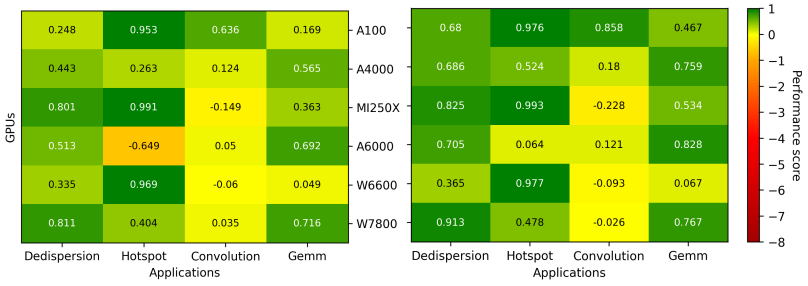
For each application, we generated two algorithms using the LLaMEA-Kernel Tuner loop of Section 7.3:

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.

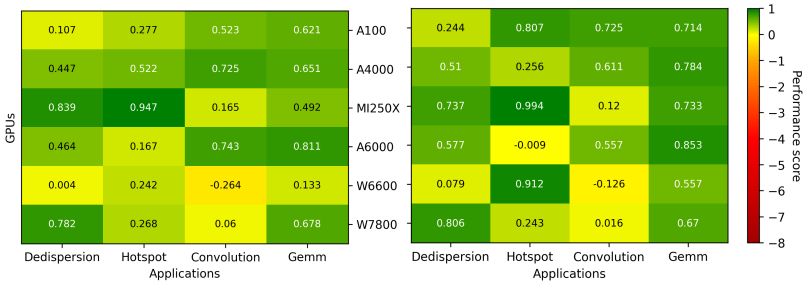
Starting with the overall performance of both versions generated for each application across all search spaces in Fig. 7.5, it is remarkable that all final versions of the generated optimization algorithms perform well in general. In these aggregate plots, each line



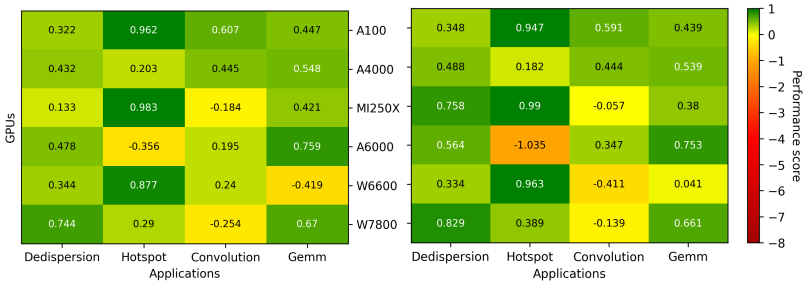

**(a)** Convolution **(b)** Convolution with extra info




**(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 $(\alpha, \beta, \delta)$ 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 $\rho p$ 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 $L$=3$p$, shake rate $s$=0.2, jump rate $q$=0.15, stagnation limit $\tau$=80, restart ratio $\rho$=0.3, $T_0$=1.0, $\lambda$=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

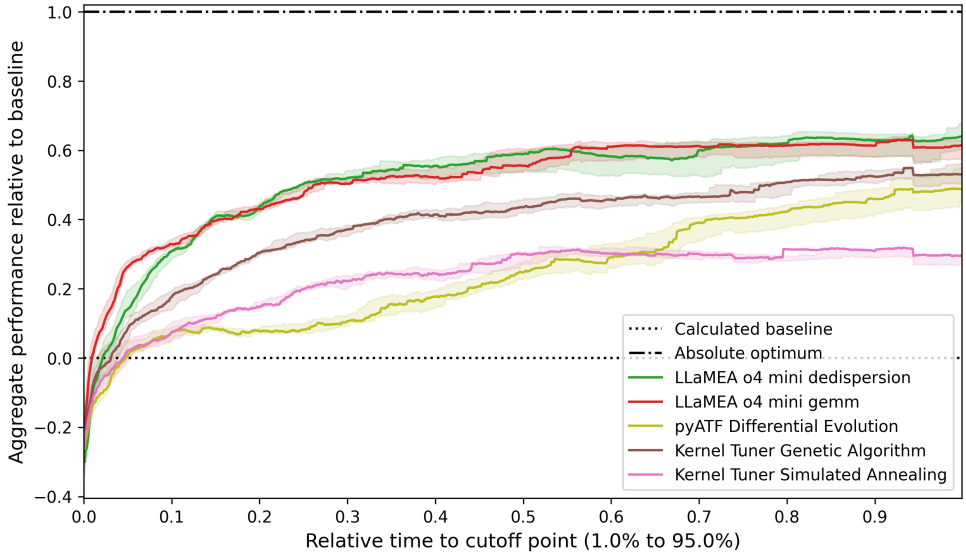
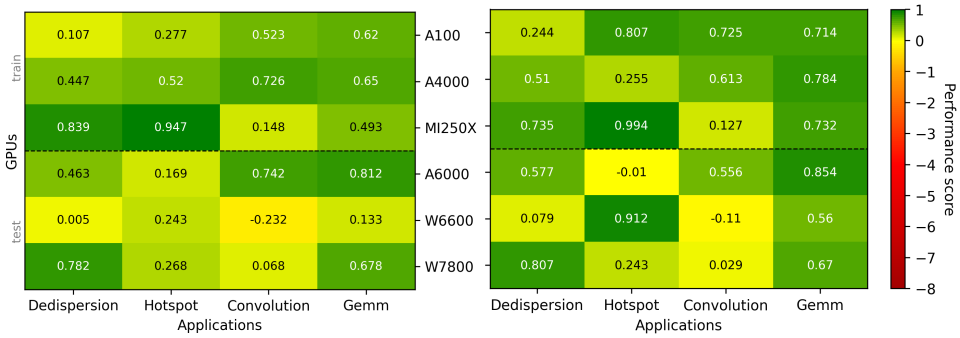



**Figure 7.7:** The aggregate performance over time over all search spaces of our generated algorithms against three human-designed optimization algorithms. Mean of 100 runs each, the shaded area marks the 95% confidence interval.

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.

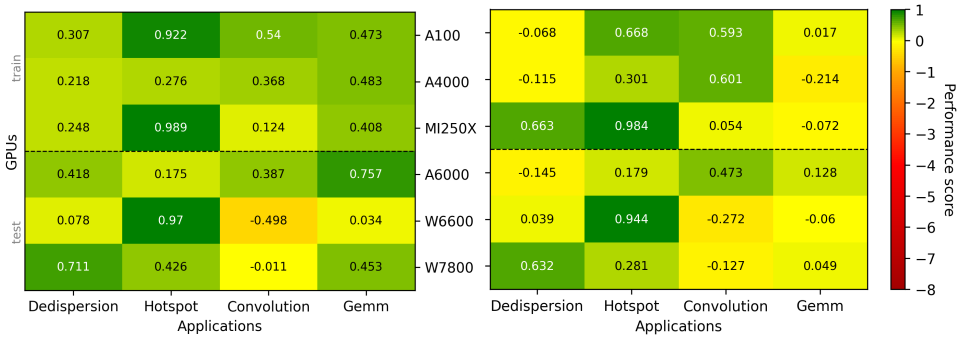
Performance stability across search spaces is also notable, as seen in Fig. 7.8. Comparing the performance of our generated algorithms against the best human-designed algorithm in the evaluation, GA, it is notable that the latter performs better on the *hotspot* application, but is otherwise outperformed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



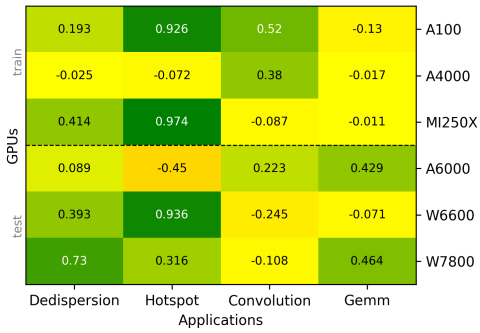

(a) Dedispersion with extra info

(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