Tuning the tuner: the art of automatic performance optimization

Willemsen, F.Q.

6 Exploring the Application of Constrained Optimization

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

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

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

6.1 Introduction

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

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

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

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

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

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

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

6.2 Background and Motivation

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


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

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

Listing 4: Example GEMM kernel in HIP/CUDA.

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

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

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

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

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

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

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

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

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

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

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

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

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

6.3 Related Work

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

6.3.1 Constraint-handling techniques

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

6.4 Design and Implementation

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

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

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

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

6.4.1 Kernel Tuner

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

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

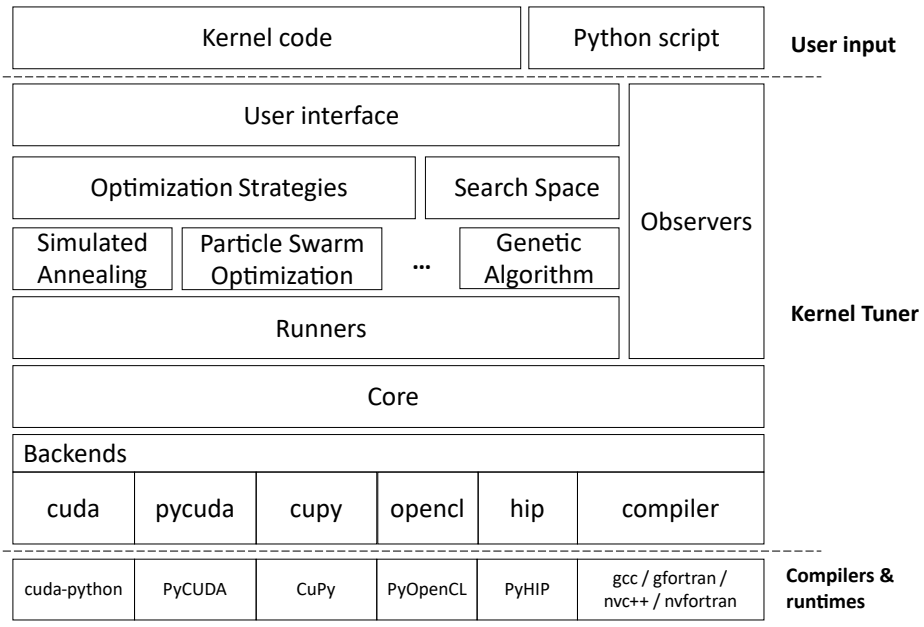


Figure 6.1: Overview of the software architecture of Kernel Tuner.

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

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

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

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

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

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

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

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

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

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

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

6.4.2 Simulated Annealing

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

Algorithm 2: Neighbor Generation for Simulated Annealing

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

Algorithm 3: Constraint-aware Neighbor Generation for Simulated Annealing

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

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

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

Algorithm 4: Constraint-aware Repair Method for PSO

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

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

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

6.4.3 Particle Swarm Optimization (PSO)

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

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

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

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

6.4.4 Genetic Algorithm

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

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

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

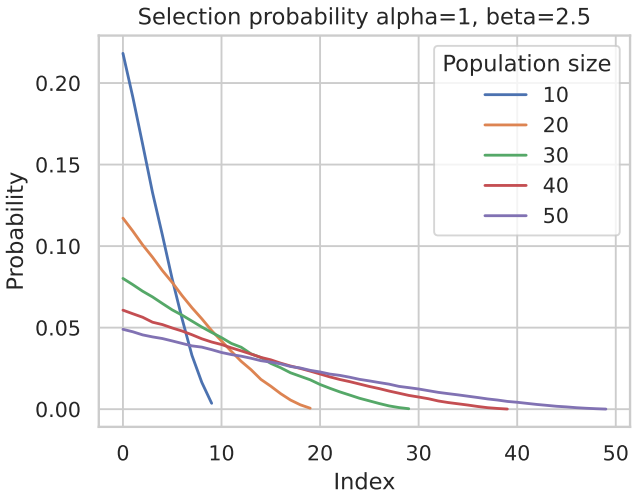


Figure 6.2: Beta-distributed relation between the index in the sorted population and the probability of being selected for crossover for different population sizes.

Algorithm 5: Constraint-aware Repair Method for GA

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

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

6.5 Evaluation

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

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

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

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

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

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

6

6.5.1 Experimental setup

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

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

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

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

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

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

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

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

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

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

The applications A, target hardware platforms H, and inputs I are the collections of A_i, H_j, and I_k of Eq. (6.1), $\mathscr{T}$ is the set of sampling points in time used to aggregate performance over time. This aggregate score enables robust comparison of optimization algorithms by capturing both the quality of the configurations found as well as the time taken to do so. As such, a difference when comparing two performance scores can indicate a difference in the quality of configurations found, the time taken to do so, or a combination of both. A score of 0.0 indicates that the performance over time is similar to the baseline, whereas a score of 1.0 indicates that the optimum is found immediately. For this evaluation section, the allocated budget for each run is equivalent to the time it takes the baseline to reach 95% of the distance between the search space median and optimum. Each optimization algorithm has been run 100 times on each search space to mitigate stochasticity.

6.5.2 Impact of Constrained Optimization for Auto-Tuning

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

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

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

Table 6.4: Hyperparameters values for the optimization algorithms.

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

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

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

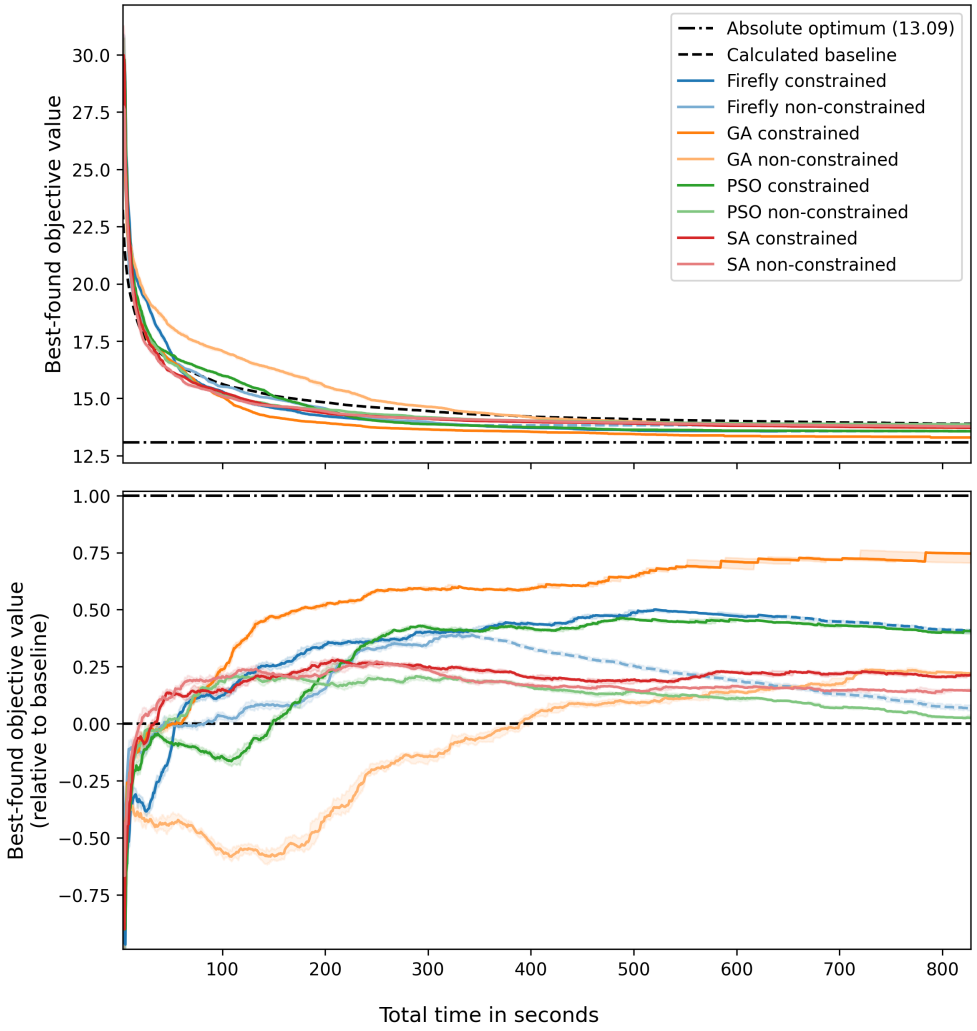


Figure 6.3: The performance over time of our constraint-aware optimization algorithms implementations and the Kernel Tuner default implementations for the GEMM kernel on the A4000.

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

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

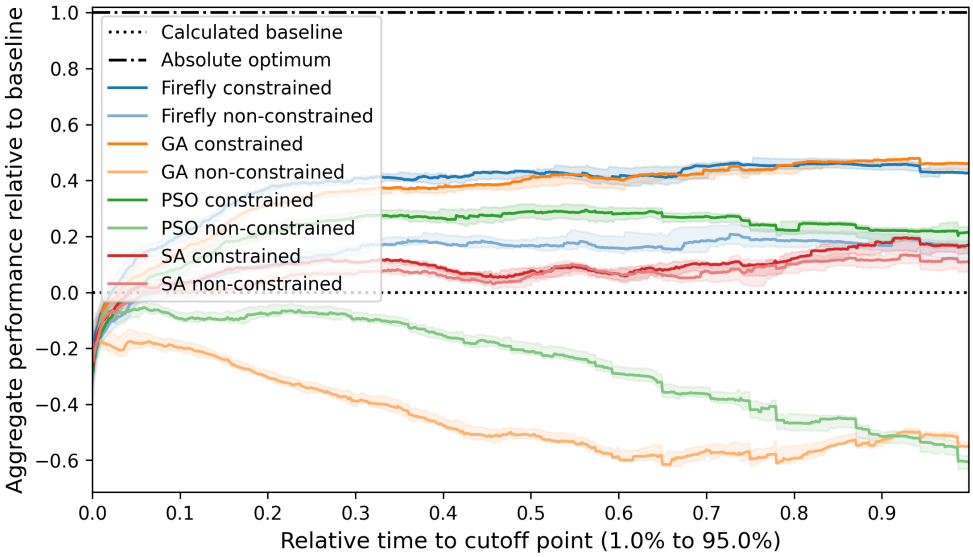
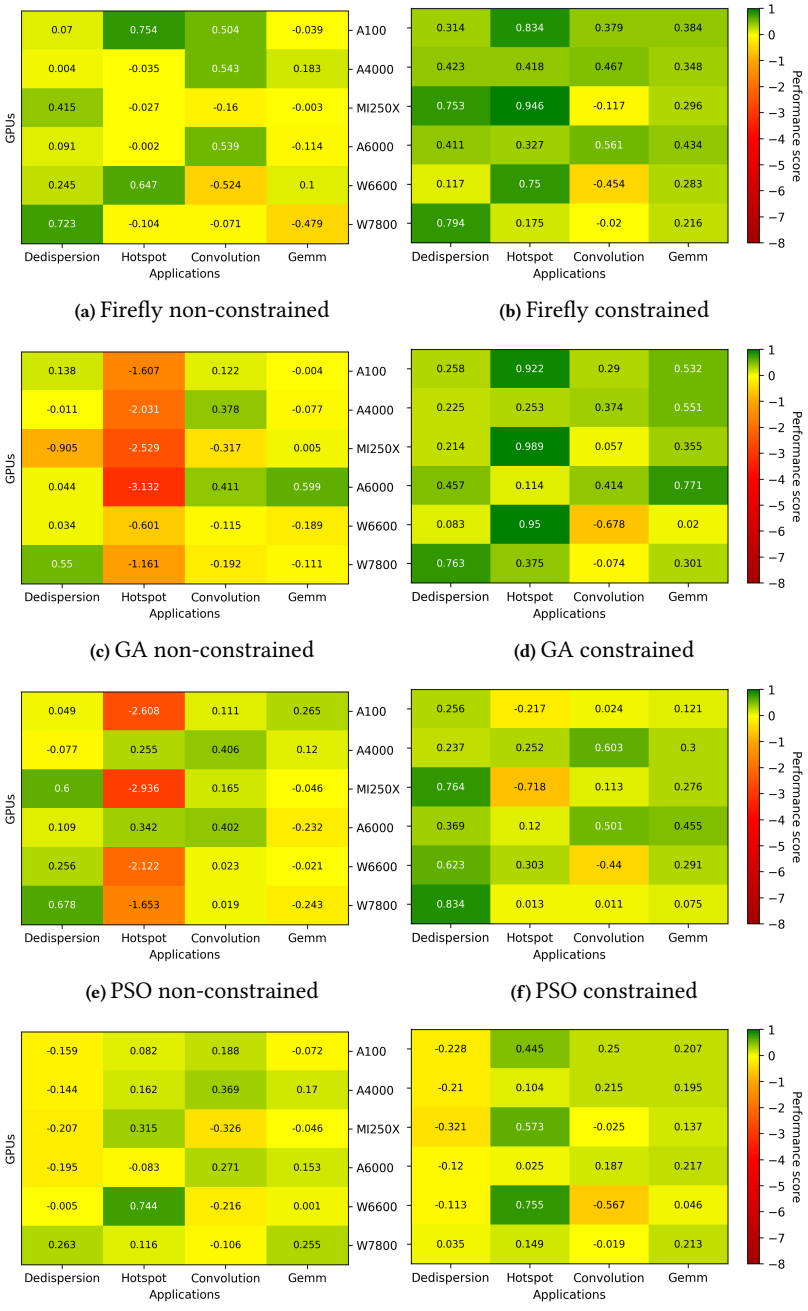


Figure 6.4: The aggregate performance over time of our constraint-aware optimization algorithms implementations and the Kernel Tuner default implementations.

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

6.5.3 Comparison with State of the Art

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



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

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

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

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

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

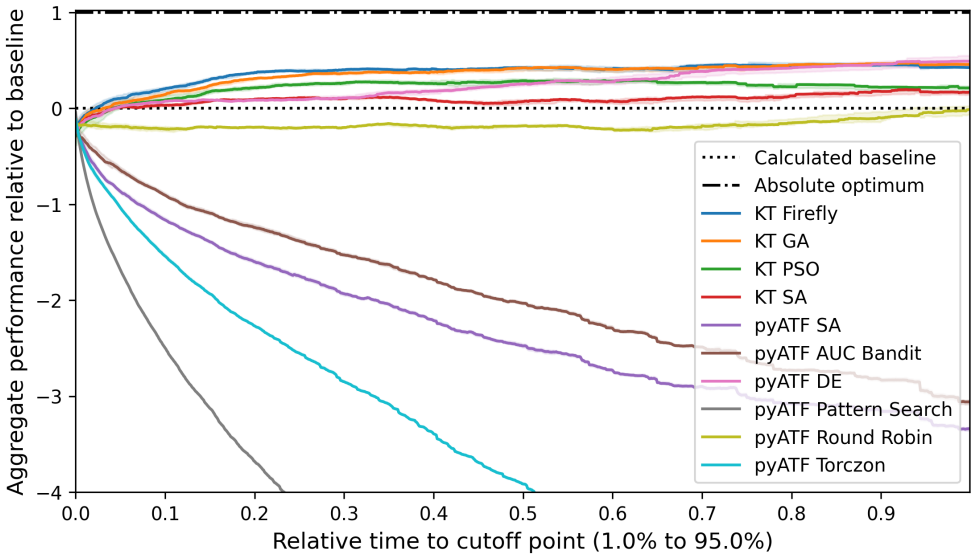
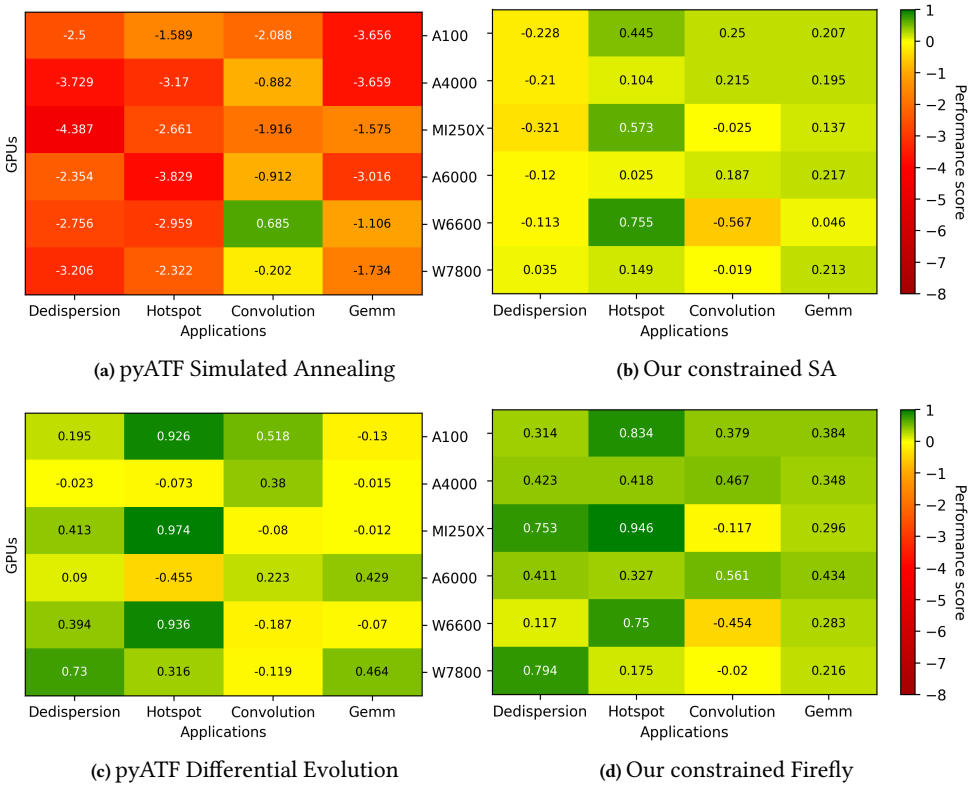


Figure 6.6: The aggregate performance of various pyATF and our constraint-aware optimization algorithms across all 24 search spaces. The plot has been cut off at -4 to improve legibility, *pyATF Torczon* and *pyATF AUC Bandit* continue to -6 and -10 respectively.

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

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

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



(a) pyATF Simulated Annealing

(b) Our constrained SA

(c) pyATF Differential Evolution

(d) Our constrained Firefly

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

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

6.6 Conclusion

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

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

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