Constraint-aware Optimization in Auto-Tuning

Floris-Jan Willemsen[1], Stijn Heldens[2], Rob V. van Nieuwpoort[1], Ben van Werkhoven[1]
[1]LIACS, Leiden University, the Netherlands {f.q.willemsen, r.v.van.nieuwpoort, b.van.werkhoven}@liacs.leidenuniv.nl
[2]Netherlands eScience Center, Amsterdam, the Netherlands, s.heldens@esciencecenter.nl

***Abstract*—Automatic performance tuning, or auto-tuning, is a key technique in high-performance computing, enabling applications to adapt to complex and evolving hardware architectures. A central challenge is the need to optimize over large discrete, constrained parameter spaces, where many candidate configurations are invalid due to hardware or software correctness constraints. Traditional evolutionary algorithms, such as Differential Evolution, Particle Swarm Optimization, and Genetic Algorithms, are not inherently constraint-aware and thus often waste computational resources evaluating invalid solutions.**

In this work, we present and evaluate constraint-aware variants of four evolutionary algorithms for auto-tuning. Through extensive experiments on a representative benchmark suite, we show that constraint-aware optimization leads to faster convergence and improved performance over unconstrained methods. Furthermore, we demonstrate that our methods outperform the pyATF methods, a state-of-the-art framework for constraint-based auto-tuning. Our results demonstrate that incorporating constraint-awareness into the optimization process significantly enhances their applicability and effectiveness in real-world auto-tuning problems. Constraint-awareness improved algorithm efficiency by $\sim$39% on average, correlated with search space sparsity. The algorithms developed in this study are publicly available as open-source contributions to the Kernel Tuner framework, facilitating future research and benefitting users.

I. Introduction

Automatic performance tuning, or auto-tuning, is a critical technique in high-performance computing (HPC), enabling developers to optimize software efficiently for specific hardware and input configurations [1, 2]. Generic auto-tuning frameworks have been created to provide an application-independent approach for users to create tunable applications, in which performance-critical parameters, or *tunable parameters*, such as the number of threads, work per thread, and data layouts, can be varied [3–6]. Auto-tuners systematically explore the vast discrete search space of code variants, generated using metaprogramming or compilation techniques, to automatically identify the optimal parameter configurations (or *solutions*) that maximize performance [7–9], energy efficiency [10], or other relevant metrics.

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. Hence, Auto-tuning frameworks 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 [5, 11], 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 such as penalty methods, constraint-specific operators, or repair mechanisms [12]. 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 paper 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 (Differential Evolution, 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 paper presents an application study of auto-tuning for performance optimization, taking real-world constraints into account. 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 constraint-aware optimization algorithms designed for automatic performance tuning as part of a generic auto-tuning framework.
- We quantify the performance impact of using constraint-aware optimization algorithms over traditional optimization algorithms for a representative benchmark [13] set of auto-tuning problems, demonstrating substantial performance improvements across various tuning scenarios.
- We present a performance comparison of our methods with pyATF [14], 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, a widely-used open-source auto-tuning framework with a broad range of features.

The remainder of this paper is organized as follows. Section II provides background on the need for constrained optimization in the auto-tuning domain. Section III reviews related work regarding constrained optimization in auto-tuning. Section IV presents the implementation of our constraint-aware optimization algorithms for auto-tuning. Section V evaluates the impact of constraint-awareness on the performance of optimization algorithms in auto-tuning. Section VI concludes.

II. 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 paper, we focus on compile-time auto-tuning where the application can be tuned as part of the development process, as opposed to run-time auto-tuning where the application is tuned while it is running in production [15].

We will use a simplified 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 1 shows a naive 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 on Line 5 a check is performed to prevent out-of-bounds array access.

Listing 1 Example GEMM kernel in HIP/CUDA.

```
1  __global__ void gemm(const float *A, const float *B,
       float *C, int M, int N, int K, float alpha, float
       beta) {
2    int row = blockIdx.y * blockDim.y + threadIdx.y;
3    int col = blockIdx.x * blockDim.x + threadIdx.x;
4
5    if (row < M && col < N) {
6      float sum = 0.0f;
7      for (int e = 0; e < K; e++) {
8        sum += A[row * K + e] * B[e * N + col];
9      }
10     C[row * N + col] = alpha * sum + beta * C[row * N
           + col];
11   }
12 }
```

For the GEMM kernel in Listing 1, 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 1 is a so-called naive kernel. A highly-optimized implementation would require extensive optimization, such as the use of tensor cores and other modifications, each introducing more tunable parameters along with more constraints. For example, to efficiently exploit the cache hierarchy and specialized memory spaces in modern architectures, we would need to introduce *loop blocking*. Loop blocking, in turn, introduces more constraints; for example, the hardware limits of specialized memory spaces such as shared memory, and the loop count needs to be a divisor of the total work assigned to a thread block or a higher-level loop-blocking scheme. 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 paper, we refer the reader to Tørring et al. [13] for a more extensive description. For a comprehensive overview of code transformation techniques, we refer to Hijma et al. [7].

III. Related Work

This section provides an overview of the application of constrained optimization in state-of-the-art auto-tuning frameworks. Table I provides an overview of the use of constrained optimization methods in auto-tuning frameworks. AUMA [16] 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 [4] maintains the full list of feasible configurations in memory but does support constraints. 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

TABLE I: 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 [16]	✗	list	K-Bagging Neural Networks	Only feasible solutions
OpenTuner [3]	✗	list	SA, PSO, Bandit, various Evolutionary Algorithms, Local Search, Random, Exhaustive	Left to the user
CLTune [4]	✓	list	SA, PSO, Neural Network, Random, Exhaustive	Only feasible solutions
ytopt [17]	✓	ConfigSpace	Bayesian Optimization	Left to the user
GPTune [18]	✓	scikit-optimize.space	Bayesian Optimization	Single constant penalty
KTT [15]	✓	chain-of-trees	Markov Chain Monte Carlo, Profile-based search, Random, Exhaustive	Only feasible solutions
ATF [5] pyATF [14]	✓	chain-of-trees	SA, Differential Evolution, Bandit, Local Search, Random, Exhaustive	Only feasible solutions
BaCO [11]	✓	chain-of-trees	Bayesian Optimization, Exhaustive	Only feasible solutions. Surrogate model for hidden constraints.
Kernel Tuner	✓	list	20 different global and local optimization algorithms, including Annealing methods, Evolutionary / Genetic methods, Swarm-based methods, and Bayesian Optimization	Single constant penalty or only feasible solutions

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 [4].

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, and how to handle the evaluation of invalid configurations, is left to the user. For example, some examples of ytopt show that infinity is returned instead of the execution time when a selected configuration does not compile successfully.

Most auto-tuning frameworks supporting user-defined constraints check the feasibility of candidate solutions as part of search space construction. ATF [5], KTT [15], BaCO [11], and pyATF [14] use chain-of-trees to store the set of valid configurations in memory.

ATF [5] 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 [14] as a new state-of-the-art framework for constraint-based auto-tuning. pyATF and ATF both include some 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 potentially expensive [19].

KTT [15] 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 is an open-source Python-based auto-tuning framework that implements many different optimization algorithms, including Differential Evolution, Particle Swarm Optimization, and Genetic Algorithm. The optimization algorithms are responsible for selecting the next code variants to be compiled and benchmarked on the GPU. Before the modifications presented in this paper, if a code variant selected by the optimizer was not valid under the constraints, Kernel Tuner applied a simple static penalty, e.g., -10^{20} [20]. In this work, we enhance this and extend Kernel Tuner with several new constraint-aware optimization algorithms to improve its performance for constraint-based auto-tuning.

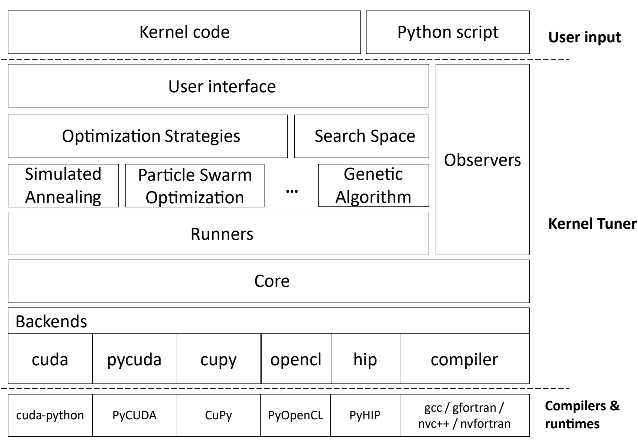


Fig. 1: Overview of the software architecture of Kernel Tuner.

IV. Design and Implementation

This section presents an overview of the design and implementation of our methods implemented in Kernel Tuner[1]. The overall auto-tuning problem can be formalized as an optimization problem which determines the optimal code variant $v^\star$ (assuming maximization) as follows:

$$v^\star = \arg\max_{v \in \mathcal{V}} f_{H_j,I_k}(A_{i,v}) \quad (1)$$

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 a search space $\mathcal{V}$.

A. 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 is shown in Figure 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 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.

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

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 [21]. Also shown in Figure 1 are the Observers which facilitate the observation of metrics besides execution time, such as the energy consumed by the GPU [22] or the numerical accuracy of the computation [23].

In Kernel Tuner, the auto-tuning search space construction problem is formalized as a Constraint Satisfaction Problem (*CSP*) [19] defined by $\mathcal{P} = (X, D, C)$, 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 $\mathcal{V} : X \to \bigcup D_i$ such that $\mathcal{V}(x_i) \in D_i$ for all i, and all constraints $c_j \in C$ are satisfied under $\mathcal{V}$. What distinguishes the auto-tuning problem from other constrained optimization problems is that C generally contains *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 $\mathcal{V}$ 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 [24]. 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 an optimized constraint satisfaction problem solver, this step can be performed with minimal impact on the total execution time [19].

The *Search Space* object in Kernel Tuner 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 four 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.

Algorithm 1 Differential Evolution

```
Require: Population X, Space V, Scaling F ∈ [0,2], Crossover CR ∈ [0,1]
 1: X ← {x_1, x_2, ..., x_N} ∈ V                          ▷ Initial population
 2: for x_i ∈ X do                                        ▷ Main evolution loop
 3:     v_i ← MUTATE(x_i, X, best)
 4:     u_i ← CROSSOVER(x_i, v_i, CR)
 5:     u_i ← REPAIR(u_i, V)
 6:     if f(u_i) ≤ f(x_i) and u_i ∉ X then
 7:         x_i ← u_i                                     ▷ Selection
 8:         if f(u_i) ≤ best then
 9:             best ← u_i                                ▷ Update best
10:         end if
11:     end if
12: end for
13: procedure MUTATE(x_i, X, best)
14:     {x_r1, x_r2, x_r3} ← select 3 random vectors from X \ {x_i}
15:     if use_best then
16:         x_r1 ← best
17:     end if
18:     {i_r1, i_r2, i_r3} ← to_indices({x_r1, x_r2, x_r3})
19:     m ← i_r1 + F · (i_r2 − i_r3)                      ▷ Apply mutation
20:     return to_values(round_and_clip(m))
21: end procedure
22: procedure CROSSOVER(x_i, v_i, CR)
23:     j_rand ← random index ∈ {1, ..., N}
24:     for j ← 1 to N do
25:         r ← random ∈ [0,1]
26:         u_i,j ← (r < CR or j = j_rand)?v_i,j : x_i,j
27:     end for
28:     return u_i
29: end procedure
30: procedure REPAIR(u_i, V)
31:     if u_i ∉ V then
32:         u_i ← get_nearest_neighbor(u_i, V)
33:     end if
34:     return u_i
35: end procedure
```

- **Index-distance**: Minimizes the sum of absolute differences between parameter value indices, returning all configurations with the lowest distance as neighbors.

The following subsections describe the implementations of our constraint-aware 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.

B. Differential Evolution

The differential evolution strategy in Kernel Tuner is quite flexible and supports many different mutation and crossover operators. Algorithm 1 shows a simplified pseudocode of the differential evolution strategy using mutation with three candidates, either three random or two random and the current best, and binomial crossover, known as the *rand1bin* or *best1bin* variants.

The algorithm starts with generating an initial population of valid candidates using Latin Hypercube Sampling (LHS) and the earlier constructed search space $\mathcal{V}$. The four main steps are mutation, crossover, repair and selection. Mutation and crossover generate a set of trial vectors $\{u_1, .., u_N\}$, which are evaluated and replace the corresponding population member $\{x_1, .., x_N\}$ if u_i outperforms x_i. Note that u_i only replaces x_i if u_i is not already in X. Also, if the population does not change at all over two consecutive generations, the population is reinitialized randomly.

The evaluation of $f(u_i)$ includes compiling and benchmarking the code variant u_i on the GPU. Kernel Tuner uses a memoization scheme to avoid evaluations of candidate solutions that have already been compiled and benchmarked.

Both mutation and crossover run the risk of generating a trial vector that violates the hard constraints on the search space. As such, the repair method is applied to each trial vector after these two steps to change the invalid trial vectors to their nearest valid neighbor in the search space, using the sum of absolute differences between parameter value indices.

The differential evolution strategy in Kernel Tuner supports all commonly-used mutation operators within differential evolution, including mutation with up to five random candidates, *currenttobest* and *randtobest* variants, which can all be combined with either binomial or exponential crossover. The non-constraint-aware version generates the initial population randomly and skips the repair step.

Algorithm 2 Constraint-aware Repair Method for PSO

```
Require: Current position x ∈ [0,1]^N, Search Space V
 1: procedure NEIGHBORS(p)
 2:     for m ∈ {'strictly-adjacent', 'adjacent', 'Hamming'} do
 3:         n ← list_neighbors(p, m) ∈ V
 4:         if |n| > 0 then
 5:             return n
 6:         end if
 7:     end for
 8:     return []
 9: end procedure
10: p ← get_params(x)                      ▷ Convert from continuous space
11: n ← NEIGHBORS(p) ∈ V
12: if |n| > 0 then
13:     n ← to_coordinates(n)              ▷ Convert to continuous space
14:     s ← Euclidian_distance(x, n)
15:     n ← sort(n, key = s)               ▷ Sort neighbors on distance to x
16:     return n[0]
17: end if
18: return []
```

C. 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 Differential Evolution, the particles in PSO are represented and move around in a continuous space.

To allow continuous optimization algorithms, such as Particle Swarm Optimization, to work on the discrete optimization problem of auto-tuning, discrete points are mapped 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 $[0, m \times \epsilon]$, 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. The continuous coordinates are converted back to discrete points by snapping to the nearest discrete point.

The position of the particle is not adjusted when converting between continuous and discrete representations. To make PSO efficient for constraint-based auto-tuning, we (i) use the search space to generate a population of only valid candidate solutions, which are then mapped to continuous coordinates as starting positions, 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 2. 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. The non-constraint-aware version generates the initial coordinates in $[0, 1]^N$ randomly and applies a single static penalty to invalid positions instead of evaluating the closest valid configuration.

D. 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 Figure 2. The advantage of this approach is that selection is biased towards better individuals independently of the magnitude of the objective function values.

After crossover, newly generated individuals may violate the constraints and therefore need to be repaired. Algorithm 3 shows the repair procedure used in GA. Reusing the NEIGHBORS procedure from PSO (Algorithm 2), 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 GA first repairs invalid solutions and then mutates if needed.

Mutation happens in exactly one parameter value, where the value is replaced with a different value for that tunable parameter, if any. The constraint-aware GA implements mutation by replacing an individual with a random valid Hamming neighbor.

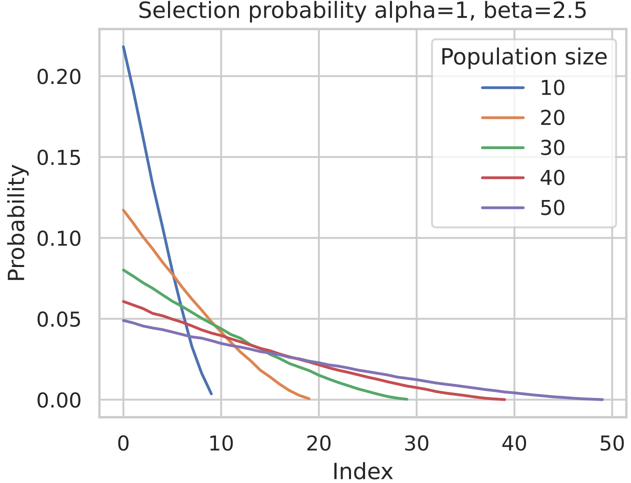


Fig. 2: Beta-distributed relation between the index in the sorted population and the probability of being selected for crossover for different population sizes.

Algorithm 3 Constraint-aware Repair Method for GA

Require: Current solution s, Search Space $\mathcal{V}$
1: **if** $s \notin \mathcal{V}$ **then**
2: $\quad n \leftarrow$ NEIGHBORS(s) $\in \mathcal{V}$
3: $\quad$ **if** $|n| > 0$ **then**
4: $\quad\quad$ **return** $random_choice(n)$
5: $\quad$ **end if**
6: **end if**
7: **return** s

V. EVALUATION

In this section, we evaluate the effectiveness of the constraint-aware optimization algorithms presented in Section IV and quantify their performance improvement. In addition, we present a comparison with pyATF, a state-of-the-art framework for constraint-based auto-tuning [14].

A. Experimental setup

For the evaluation, we focus on six different GPU models available in the DAS-6 [25] and LUMI [26] supercomputers. The GPU specifications are listed in Table II. On DAS-6, we use Rocky-8 Linux 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 Kernel Tuner version 1.3.1 and compared against pyATF version 0.0.9.

We will evaluate our approach using the BAT benchmark suite [13] 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

TABLE II: 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 (GFLOPS/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 III: Overview of the basic characteristics of the real-world applications.

Name	Cartesian size	Constrained 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

science, and linear algebra, respectively. The characteristics of these applications are shown in Table III. 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 [27].

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 II. This tunable GEMM kernel originates from CLBlast [28], 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 II, 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 [29]. This methodology provides a systematic approach to comparing optimization algorithms across auto-tuning search spaces, reflecting the actual total time spent (including resolving the search space) as this is most relevant for real-world usage. 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* $\mathcal{P}$ 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.

$$\mathcal{P}(\mathcal{F}, A, H, I) = \frac{1}{|\mathcal{T}|} \sum_{t \in \mathcal{T}} \frac{\sum_{A_i \in A} \sum_{H_j \in H} \sum_{I_k \in I} \mathcal{P}(\mathcal{F}, A_i, H_j, I_k)_t}{|A||H||I|} \tag{2}$$

The applications A, target hardware platforms H, and inputs I are the collections of A_i, H_j, and I_k of Equation (1), $\mathcal{T}$ 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.

B. 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 Differential Evolution (DE), Particle Swarm Optimization (PSO), Firefly Algorithm, and Genetic Algorithm (GA) with and without

TABLE IV: Hyperparameter values for the optimization algorithms.

Algorithm	Hyperparameter	Values
Differential Evolution (*DE*)	popsize	16
	differential weight	0.7
	crossover rate	0.6
	method	best1bin
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

the modifications for constrained optimization presented in Section IV.

The hyperparameters of the optimization algorithms are shown in Table IV, with the hyperparameters tuned as per the extended tuning method of [30]. Both the constrained and nonconstrained versions of the optimization algorithms use the same hyperparameters. The *popsize* parameter in DE is multiplied by the number of dimensions to get the actual population size. 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 3 compares this over time in two plots for the Dedispersion kernel on the Nvidia A6000. 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 84.218 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 V-A, 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 Figure 3, we can see that most algorithms start with negative scores, meaning that at the start, the algorithms perform worse than the calculated baseline. It can be seen in Figure 3 that each constraint-aware optimization algorithm outperforms its non-optimized counterpart by a significant margin. It is particularly noteworthy that for PSO, Firefly, and GA, the performance of the non-constrained variants approaches that of the random search baseline for the first half of the tuning time.

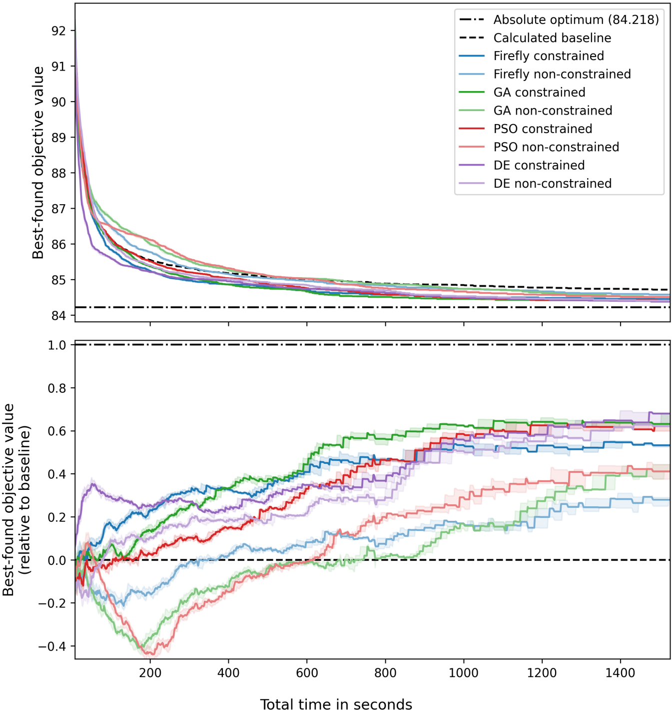


Fig. 3: The performance over time of our constraint-aware optimization algorithms implementations and the Kernel Tuner default implementations for the Dedispersion kernel on the Nvidia A6000.

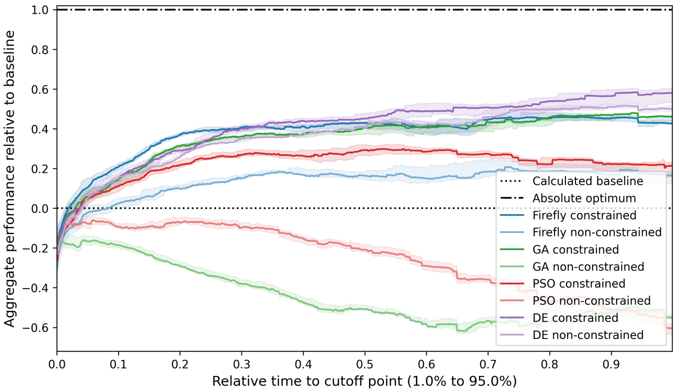


Fig. 4: The aggregate performance over time of our constraint-aware optimization algorithms implementations and the Kernel Tuner default implementations.

To evaluate the overall performance, Figure 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 Differential Evolution (*DE*) has a minor gain with 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, PSO by 0.478, DE by 0.047, and Firefly by 0.245, for an average improvement of 0.393. This can be interpreted as these algorithms finding the

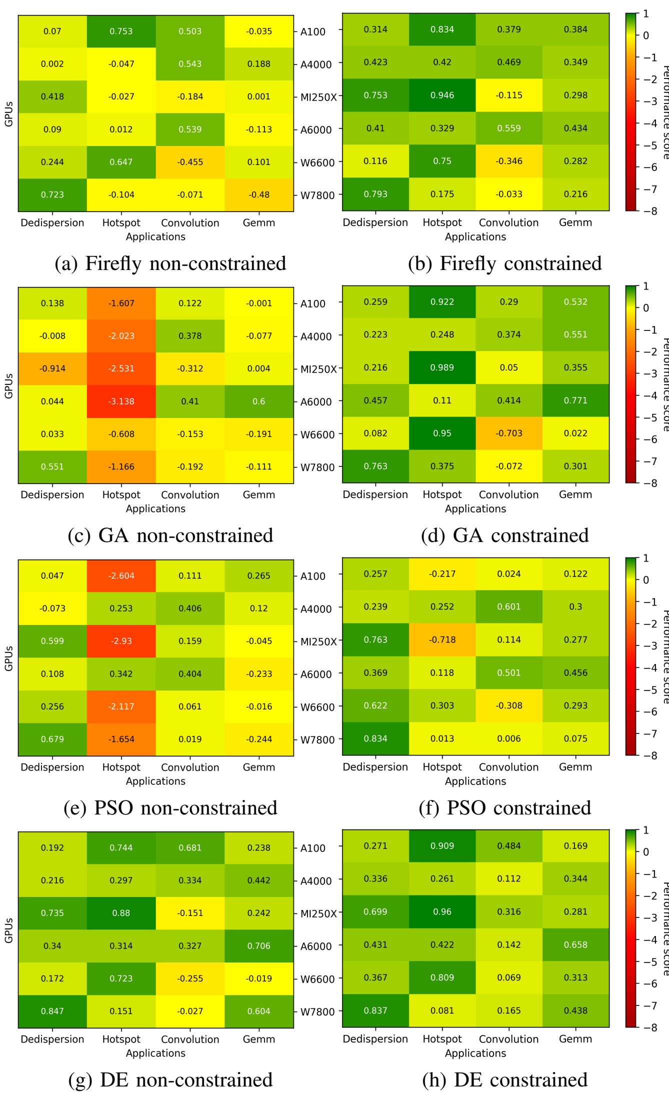


Fig. 5: Impact of constraint-awareness on optimization algorithm performance per search space.

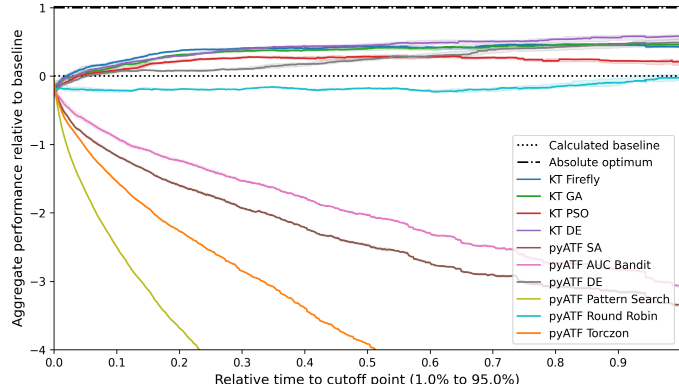


Fig. 6: The aggregate performance of various pyATF and our constraint-aware optimization algorithms across all 24 search spaces. The plot has been cut off at -4 to improve legibility, *pyATF Torczon* and *pyATF AUC Bandit* continue to -6 and -10 respectively.

same configuration in $\sim 39\%$ less time, finding $\sim 39\%$ better performing configurations in the same time, or a combination.

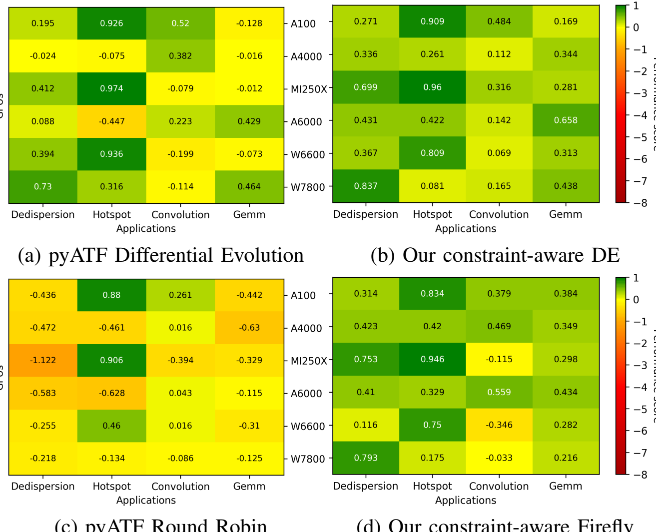
Finally, we can compare the performance score per search space between our constraint-aware and the original versions for specific optimization algorithms to validate our findings, shown in Figure 5. As would be expected when the performance improvement is due to constraint-awareness, the performance difference is greatest between the sparsest search spaces, especially *hotspot* and to a lesser extent *GEMM*, as per Table III. This is particularly noticeable for GA and PSO, where the performance difference between both versions was also the largest in Figure 4.

C. Comparison with State of the Art

In this subsection, we evaluate the performance of our constraint-aware optimization algorithms against pyATF [14], 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 [19], 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 Kernel Tuner does. To ensure a fair comparison of only the optimization algorithms in both frameworks, we have used Kernel Tuner's cost function for pyATF's methods as well, 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 in pyATF with default hyperparameters; 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 Figure 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 Figure 4. Quantifying this with the performance score, the average score of our optimization algorithms is 0.342, 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 constraint-aware optimization algorithms we introduce in this paper. This is shown in Figure 7 for *Differential Evolution*, which is present in both frameworks, as well as for the next best-performing algorithms. In the latter case, our constraint-aware Firefly (Figure 7d) outperforms pyATF Round Robin (Figure 7c) on all but two of the 24 search spaces. The



(a) pyATF Differential Evolution (b) Our constraint-aware DE

(c) pyATF Round Robin (d) Our constraint-aware Firefly

Fig. 7: Performance difference per search space for *Differential Evolution* and the next best-performing algorithms between pyATF and the constraint-aware implementations presented in this paper.

pyATF implementation of *Differential Evolution* (Figure 7a) is outperformed by our DE implementation (Figure 7a) on the majority of search spaces as well.

VI. 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 optimization. In this work, we addressed this challenge by integrating constraint-handling strategies into four widely used evolutionary optimization algorithms, Differential Evolution, 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 ($\sim$ 39%) compared to their non-constraint-aware counterparts, correlated with search space sparsity. We observed faster convergence, more efficient exploration of the feasible search space, and higher-quality solutions across a broad range of auto-tuning search spaces. Furthermore, we showed that our methods outperform the methods in the state-of-the-art pyATF framework, 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 framework, lowering the barrier to adoption and enabling reproducibility.

Acknowledgment

The CORTEX project has received funding from the Dutch Research Council (NWO) in the framework of the NWA-ORC Call (file NWA.1160.18.316). The ESiWACE3 project has received funding from the European High Performance Computing Joint Undertaking (JU) under grant agreement No 101093054.

References

[1] P. Balaprakash, J. Dongarra *et al.*, “Autotuning in High-Performance Computing Applications,” *Proc. IEEE*, 2018.
[2] B. van Werkhoven, W. J. Palenstijn *et al.*, “Lessons learned in a decade of research software engineering GPU applications,” in *International Conference on Computational Science (ICCS)*, 2020.
[3] J. Ansel, S. Kamil *et al.*, “OpenTuner: An extensible framework for program autotuning,” in *Parallel Architectures and Compilation Techniques (PACT)*, 2014.
[4] C. Nugteren and V. Codreanu, “CLTune: A generic auto-tuner for OpenCL kernels,” in *Multicore/Many-core Systems-on-Chip (MCSoC)*, 2015.
[5] A. Rasch and S. Gorlatch, “ATF: A generic directive-based auto-tuning framework,” *Concurr. Comput.*, 2018.
[6] B. van Werkhoven, “Kernel Tuner: A search-optimizing GPU code auto-tuner,” *Future Gener. Comput. Syst.*, 2019.
[7] P. Hijma, S. Heldens *et al.*, “Optimization Techniques for GPU Programming,” *ACM Comput. Surv.*, 2023.
[8] M. Frigo and S. G. Johnson, “Fftw: An adaptive software architecture for the fft,” in *Acoust. Speech Signal Process.*, 1998.
[9] R. C. Whaley, A. Petitet *et al.*, “Automated empirical optimizations of software and the ATLAS project,” *Parallel Comput.*, 2001.
[10] R. Schoonhoven, B. Veenboer *et al.*, “Going green: Optimizing GPUs for energy efficiency through model-steered auto-tuning,” *PMBS*, 2022.
[11] E. O. Hellsten, A. Souza *et al.*, “BaCO: A fast and portable bayesian compiler optimization framework,” in *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2024.
[12] C. A. Coello Coello, “Theoretical and numerical constraint-handling techniques used with evolutionary algorithms: a survey of the state of the art,” *Comput. Methods Appl. Mech. Eng.*, 2002.
[13] J. O. Tørring, B. van Werkhoven *et al.*, “Towards a benchmarking suite for kernel tuners,” in *International Workshop on Automatic Performance Tuning (iWAPT)*, 2023.
[14] R. Schulze, S. Gorlatch *et al.*, “pyATF: Constraint-based auto-tuning in python,” in *Compiler Construction (CC)*, 2025.
[15] F. Petrovič and J. Filipovič, “Kernel tuning toolkit,” *SoftwareX*, 2023.
[16] T. L. Falch and A. C. Elster, “Machine learning based auto-tuning for enhanced OpenCL performance portability,” in *International Workshop on Automatic Performance Tuning (iWAPT)*, 2015.
[17] X. Wu, P. Balaprakash *et al.*, “Ytopt: Autotuning Scientific Applications for Energy Efficiency at Large Scales,” *Concurr. Comput.*, 2024.
[18] Y. Liu, W. M. Sid-Lakhdar *et al.*, “GPTune: Multitask Learning for Autotuning Exascale Applications,” in *Principles and Practice of Parallel Programming (PPoPP)*, 2021.
[19] F. J. Willemsen, R. V. van Nieuwpoort *et al.*, “Efficient construction of large search spaces for auto-tuning,” in *International Conference on Parallel Processing (ICPP)*, 2025.
[20] R. A. Schoonhoven, B. van Werkhoven *et al.*, “Benchmarking optimization algorithms for auto-tuning gpu kernels,” *Trans. Evol. Comput.*, 2022.
[21] M. Lurati, S. Heldens *et al.*, “Bringing auto-tuning to HIP: Analysis of tuning impact and difficulty on AMD and Nvidia GPUs,” in *European Conference on Parallel Processing*, 2024.
[22] S. v. d. Vlugt, L. Oostrum *et al.*, “Powersensor3: A fast and accurate open source power measurement tool,” in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2025.
[23] S. Heldens and B. van Werkhoven, “Accuracy-Aware Mixed-Precision GPU Auto-Tuning,” *IEEE Trans. Parallel Distrib. Syst. (accepted for publication)*, 2026.
[24] F. J. Willemsen, R. van Nieuwpoort *et al.*, “Bayesian Optimization for auto-tuning GPU kernels,” in *PMBS*, 2021.
[25] H. Bal *et al.*, “A medium-scale distributed system for computer science research: Infrastructure for the long term,” *Computer*, 2016.
[26] K. Koski, P. Manninen *et al.*, *Fifty Years of High-Performance Computing in Finland*, 2023.
[27] A. Sclocco, S. Heldens *et al.*, “AMBER: a real-time pipeline for the detection of single pulse astronomical transients,” *SoftwareX*, 2020.
[28] C. Nugteren, “CLBlast: A tuned OpenCL BLAS library,” in *Int. Workshop OpenCL*, 2018.
[29] F. J. Willemsen *et al.*, “A methodology for comparing optimization algorithms for auto-tuning,” *Future Gener. Comput. Syst.*, 2024.
[30] F. J. Willemsen, R. V. van Nieuwpoort *et al.*, “Tuning the tuner: Introducing hyperparameter optimization for auto-tuning,” in *IEEE eScience*, 2025.