Efficient Construction of Large Search Spaces for auto-tuning

ICPP 2025 Submission #**NaN** – Confidential Draft – Do NOT Distribute!!

Anonymous Author(s)

Abstract

Creating efficient high-performance programs such as GPU applications involves a vast number of choices, such as thread configurations, memory access patterns, and algorithmic variations, all of which can significantly impact performance. Auto-tuning is used to optimize the performance, accuracy, and energy efficiency of such programs to select the best configurations from a vast search space. However, the construction of this search space can be a bottleneck at the start of the tuning process due to the large number of possible combinations and the complex constraints applied to each of these. Real-world applications have been encountered where the search space construction takes minutes to hours or even days.

This work addresses this challenge by leveraging Constraint Satisfaction Problem (CSP) solvers to efficiently construct auto-tuning search spaces. We introduce several key optimizations to enhance solver efficiency, substantially reducing the overhead of search space construction while maintaining flexibility and scalability, and provide easy-to-use implementations to the CSP and auto-tuning communities. Evaluation on real-world applications demonstrates that our optimized solver improves search space construction performance by four orders of magnitude over naive solving and is one to two orders of magnitude faster than the current state-of-the-art. This results in a tangible improvement to the overall auto-tuning process, enabling the exploration of previously unattainable problem scales in auto-tuning and related domains.

CCS Concepts

• **Computing methodologies** → *Discrete space search.*

Keywords

Constraint solving, Auto-tuning, Search spaces, CSP, HPC

1 Introduction

Automatic performance tuning, or auto-tuning [5], is a commonly applied technique in high-performance computing for optimizing programs towards a particular hardware architecture. Auto-tuning allows developers to automate the process of exploring the myriad of implementation choices that arise in performance optimization, such as the number of threads, tile sizes used in loop blocking, and other code optimization parameters [18]. Many well-known examples of auto-tuned high-performance libraries and applications exist, such as FFTW [14] for Fast Fourier Transforms, or ATLAS [38] for linear algebra. At the heart of the auto-tuning method is a *search space* of functionally-equivalent *code variants* that is explored by an *optimization algorithm*. These code variants can be generated by a compiler, or using metaprogramming techniques, such as application-specific code generators or function templates.

Together these code variants constitute vast search spaces that are infeasible to search by hand [25, 30, 32] and would have to be searched over and over again as the application is executed on different hardware or different input data sets and sizes [19, 24, 27, 37]. Construction of the auto-tuning search space, used for enumerating and sampling different code variants, is a crucial factor in the performance of auto-tuning as search space sizes increase, and has therefore received a lot of attention recently [16, 28, 29, 31].

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

In modern applications, where search spaces may contain millions or even billions of configurations, constructing the search space can take minutes to days, making search space construction a major bottleneck [15, 16, 34]. This is problematic when auto-tuning for a single target, but even more prohibitive on a diverse set of target input data and hardware, such as a BLAS library.

To this end, Rasch et al. [28] have introduced a method, referred to as *chain-of-trees*, specifically designed for the purpose of efficiently constructing search spaces for constraint-based auto-tuning. The chain-of-trees approach starts with identifying groups of interdependent parameters. Two parameters are interdependent if they both occur in the syntax tree of the same constraint descriptor. For each parameter group, a tree is constructed that encodes all possible combinations of interdependent parameter values. Finally, the trees are linked together to form a chain-of-trees. The chain-of-trees method is widely considered to be state-of-the-art and has been integrated into several auto-tuning frameworks, including ATF [29], BaCO [16], PyATF [31], and KTT [26].

In this paper, instead of adopting a customized solution for constraint-based auto-tuning, such as chain-of-trees, we investigate the use of methods with a more robust mathematical foundation. In particular, we show that the problem of search space construction in constraint-based auto-tuning can be automatically reduced to a Constraint Satisfaction Problem (CSP). To enable the use of CSP in a state-of-the-art auto-tuner, we employ various techniques, including run-time compilation to translate user-defined constraint functions to representations that can be used directly in existing CSP solvers, as well as major improvements to the performance of such solvers. We evaluate the CSP and chain-of-trees methods on a wide variety of search spaces and scenarios, including the search spaces used by Rasch et al. [29].

'ICPP 2025', September 8–11, 2025, San Diego, USA

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

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

Listing 1: **Example Hotspot kernel in HIP/CUDA.**

Our work has been integrated into the auto-tuning framework Kernel Tuner [36] and a Python-based CSP solver named python-constraint, both existing open-source projects with a substantial number of users, to benefit both communities.

The rest of this paper is structured as follows. Section 2 provides a high-level introduction to the role of constraints in auto-tuning. Section 3 discusses related work. Section 4 describes the design and implementation of our method. In Section 5, we evaluate the efficiency and scalability of our optimized CSP-based approach against various state-of-the-art solutions, and Section 6 concludes.

2 Constraint-based Auto-tuning

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

We will use the Hotspot kernel from the BAT [34] benchmark suite of tunable kernels as an example. This Hotspot kernel, adapted from the Rodinia Benchmark suite [11], simulates heat dissipation in a microprocessor based on the processor's architectural floor plan, thermal resistance, ambient temperature, and simulated power currents. Listing 1 shows the Hotspot kernel code in HIP/CUDA, simplified for readability by removing bound checks.

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

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

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

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

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

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

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

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

Listing 2: **Example of constraints specification in different tuners.**

specific combination of tunable parameter values to be considered a *valid* candidate solution, or code variant, in the search space.

The constraint defined in Listing 2 only involves the thread block dimensions. However, the fully optimized version of the Hotspot kernel contains many more tunable parameters and constraints. For example, the number of output elements computed by each thread can be varied in both the x- and y-dimensions, introducing two more tunable parameters. Another tunable parameter (`sh_power`) controls whether or not to cache the `Power` values in *shared memory*. Together, these parameters form another constraint, namely `(thread_block_x * work_per_thread_x) * (thread_block_y * work_per_thread_y) * sh_power * 4 <= max_shared_memory_per_block` to avoid exceeding the maximum amount of shared memory allowed per block in bytes. The full Hotspot kernel is even more complex and also implements temporal tiling, partial loop unrolling, and double buffering of the temperature field in shared memory, resulting in several more complex constraints. For further specification of the kernel see [34].

3 Related Work

Table 1 shows an overview of support for specifying constraints, as well as the method used for search space construction in related auto-tuning frameworks. As we can see, some frameworks rely on brute force search space construction, *ytopt* and *GPTune* use `ConfigSpace` and `scikit-optimize.space` respectively, while chain-of-trees is the most commonly used. We will briefly discuss the pros and cons of these different approaches.

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

Table 1: **Overview of constraint support and search space construction methods in related work and this work (Kernel Tuner). ⋆ While ytopt and GPTune are actively maintained, dependencies *ConfigSpace* and *scikit-optimize* are not.**

Tuner	Open Source	Actively developed	Constraints API	Search Space Construction
AUMA [13]	✓	✗	n/a	external
CLTune [25]	✓	✗	C++	brute-force
OpenTuner [2]	✓	✗	n/a	brute-force
ytopt [40]	✓	✓⋆	Python	ConfigSpace
GPTune [21]	✓	✓⋆	Python	scikit-optimize.space
KTT [26]	✓	✓	C++	chain-of-trees
ATF [28]	✓	✓	C++	chain-of-trees
BaCO [16]	✓	✗	JSON	chain-of-trees
PyATF [31]	✓	✓	Python	chain-of-trees
Kernel Tuner	✓	✓	Python	CSP solver

the search space size and number of constraints increase. For various auto-tuning applications, the number of valid configurations in the search space is orders of magnitudes smaller than the Cartesian product size, causing the vast majority of generated and evaluated combinations to be discarded, wasting time and resources.

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

Rasch et al. [28] introduced the chain-of-trees structure to represent constrained search spaces in constraint-based auto-tuning. Parameters are grouped based on interdependencies in the constraint functions, and each group is represented by a tree that encodes valid parameter combinations. These trees are then linked sequentially, forming a chain. This structure can reduce redundancy by reusing shared subtrees, which may lead to lower memory usage compared to flat representations. Independent parameters are handled as single-parameter trees.

In Section 4, instead of adopting a customized solution for constraint-based auto-tuning such as chain-of-trees, we investigate the feasibility of using established methods with a robust mathematical foundation for efficient search space construction.

4 Application and Optimization of CSP Solvers

In this section, we will discuss the design and implementation details of our novel method for efficiently constructing large search spaces for constraint-based auto-tuning. Section 4.1 examines various constraint-solving techniques to formalize the relation with auto-tuning and find a robust solver best suited to the problem context. Following this, we automatically optimize accessible user-defined constraints in Section 4.2. Next, the solver is optimized in Section 4.3 to make it efficient in the auto-tuning domain. Finally, the representation and application of the resulting search space in auto-tuning frameworks are detailed in Section 4.4.

4.1 Using Constraint Solvers in Auto-tuning

The search space construction problem, where parameter values and constraints must be resolved to all valid combinations, can generally be encoded as a Boolean Satisfiability Problem (SAT) [8], Satisfiability Modulo Theories (SMT) [6], or Constraint-Satisfaction Problem (CSP) [10]. Among SAT, SMT, and CSP, the technique closest to auto-tuning search space construction is CSP. While SAT deals with Boolean variables and SMT extends SAT with theories like arithmetic or arrays, CSP natively supports multi-valued variables and more general and complex constraint types, making it more expressive and better suited for modeling the relationships found in auto-tuning problems.

We can formalize the auto-tuning search space construction problem as a CSP 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 $\text{scope}(c_j) \subseteq X$.

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 $\mathcal{V}$. The overall auto-tuning objective is to determine the optimal configuration as $v^\star = \arg\max_{v \in \mathcal{V}} f_{H_j, I_k}(A_{i,v})$, where we have an application A_i on a hardware platform H_j for an input dataset I_k to maximize performance $f_{H_j, I_k}(A_i)$ over the code variants in $\mathcal{V}$.

In addition, there are practical considerations when it comes to choosing a solver to build on. As we will implement our solver in the Python-based Kernel Tuner, the solver should be deployable in a Python environment. Notable options include *OR-Tools* [22] and the Z3 solver in *PySMT* [33], which provide highly expressive modeling capabilities and efficient solving algorithms. However, most solvers aim to find any solution, rather than all solutions, as required in the case of auto-tuning search space construction. To obtain all solutions, such solvers must iteratively find a solution, add this solution as an additional constraint, and look for the next solution until there are no solutions left [9]. If there are many solutions, as is commonly the case with auto-tuning problems, this can have a substantial negative impact on performance, as will be shown in Section 5.2. A notable exception to this is *python-constraint* [23], as this is a CSP-based Python package capable of finding all solutions, which we will use as the basis of our implemented method.

4.2 Parsing Constraints

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

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

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

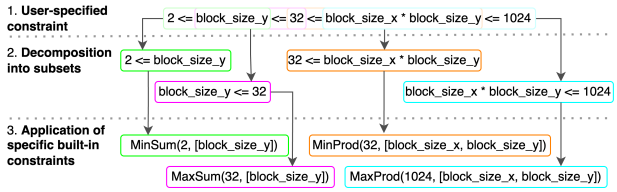

Figure 1: **The optimization of a constraint via the parsing pipeline.**

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

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

In particular, the automatic reduction of constraints can be important in scalability and efficiency in practice, as users unfamiliar with the intrinsics of constraint solving, such as the users of auto-tuning frameworks, might write sub-optimal constraints in practice. For example, consider a Kernel Tuner constraint `2 <= block_size_y <= 32 <= block_size_x*block_size_y <= 1024` like in Listing 2, where `block_size_x`, `block_size_y` are tunable parameters with numerical values. Constraints can not be evaluated until values for the involved parameters are at least partially resolved, resulting in subpar performance in the case of compound statements like the given example, as it depends on the resolution of both parameters as-is. This can be improved by automatically breaking down the constraint into multiple constraints with fewer involved variables where possible. For the given example this can be decomposed as shown in Step 2 of Figure 1, which allows partially resolved values for `block_size_x` or `block_size_y` to discard configurations not meeting the constraint earlier in the construction process.

In addition, this automatic reduction enables the application of specific constraints, as is the case with the example, which can be represented with specific constraints as shown in Step 3 of Figure 1. As will be further discussed in Section 4.3.2, the application of specific constraints can preemptively exclude values through preprocessing, resulting in an even more efficient construction.

4.3 Implementation of Optimizations

To obtain the level of performance required to construct auto-tuning search spaces efficiently, we implement several key improvements in various areas of the *python-constraint* package: algorithmically (Section 4.3.1), by extending constraints (Section 4.3.2), engineering (Section 4.3.3), and by tailoring output formats (Section 4.3.4).

4.3.1 Algorithm. We select and optimize a backtracking solver optimized for finding all solutions rather than any solution. This algorithm maintains a dictionary of variable assignments and uses a stack-based approach to implement iterative backtracking, avoiding recursive function calls. Variables are selected dynamically using a combination of the Minimum Remaining Values (MRV) and Degree heuristics, prioritizing those with fewer remaining values and higher connectivity. For each selected variable, domain values are checked against the constraints. If a constraint is violated, the algorithm backtracks by restoring previous states from a stack until all possibilities are explored. We optimize this algorithm further by sorting the variables on the number of internal constraints, making it faster to find unassigned variables, and by reducing the number of sorts required.

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

However, not all constraints can be expressed as built-in constraints, for example when using an operation that is not as common. Such cases are parsed to *Function* constraints, which we have optimized by employing function rewriting and dynamic runtime compilation, as the one-off expense of compilation to bytecode is offset by the many times a *Function* constraint is usually executed.

4.3.3 Employing C-extensions. In general, C and similar languages outperform Python in terms of execution speed [1, 41]. To attain this level of performance without losing the flexibility and user-friendliness of Python [3], we employ C-extensions. We transpile the codebase from Python to C-code using Cython [7], which is then compiled into Python-importable C-extensions. We added type hints where possible to aid in compilation. Binaries are precompiled for Linux, macOS, and Windows for all supported Python versions.

4.3.4 Output Formats. We implement various output formats to avoid expensive rearrangements to different formats. Expensive rearrangement of the structure in which solutions are output by the solver is mitigated by providing output formats that are close to the internal representation, further described in Section 4.4.

4.4 Search space Representation

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

After the search space construction, optimization algorithms use the information obtained in the construction step to select configurations. Instances of this are obtaining the true bounds of

the search space to use balanced initial sampling methods or the selection of valid neighbors that have not been evaluated yet.

This brings up an important benefit of our method compared to the other methods in Section 3, as search space characteristics such as the true parameter bounds, which can help optimization algorithms navigate the space more efficiently and enable fairly distributed sampling methods such as Latin Hypercube Sampling [39], are not resolved with dynamic approaches as they require a fully enumerated space of configurations. Furthermore, randomized sampling is inherently biased to the sparser parts of the chain-of-trees, although this has been addressed by BaCO [16]. Moreover, selecting valid neighbors of configurations as extensively used by various optimization algorithms in Kernel Tuner is potentially expensive.

Instead, we fully resolve the search space before starting the tuning process, with a minimal impact on the total execution time, to incorporate the full information of the search space in the initial sampling and optimization algorithms. As operations such as sampling and finding valid neighbors are commonly used in auto-tuning, it can be useful to provide an abstract representation of the search space that implements these operations, providing various views and mappings on the configurations in the search space.

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

5 Evaluation

In this section, we evaluate the advancements presented in Section 4 to determine their scalability and performance impact. First, we discuss how we compare against the current state-of-the-art solvers in Section 5.1. We then evaluate the solvers on a large collection of synthetically generated search spaces with varying characteristics to assess scalability differences in Section 5.2. Following this, we evaluate the solvers on a variety of real-world applications to assess the performance improvement in Section 5.3. Finally, we validate the practical impact of our method on the entire auto-tuning pipeline in Section 5.4.

The evaluations in this work are performed on the sixth generation DAS VU-cluster [4] using an NVIDIA A100 GPU node. The GPU is paired with a 24-core AMD EPYC-2 7402P CPU, 128 GB of memory, and running Rocky Linux 4.18. For all tests performed, the results of each solver were validated against a brute-forced solution of the search space.

5.1 Comparison against state-of-the-art

To provide additional reference on the performance in this evaluation, we compare the results to the state-of-the-art in auto-tuning search space construction, the chain-of-trees of Auto-Tuning Framework (ATF). ATF has two independent implementations, in C++ [28] and Python [31], both of which we use in this evaluation to compare our method. The C++ version available as of August 2024 with Python bindings is used and denoted as *ATF* in the results. The Python version called *pyATF* is used at version 0.0.9, the latest version at the time of writing.

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

In addition, we compare against PySMT at version 0.9.6 using the aforementioned Z3 solver, developed by Microsoft for software verification and analysis [12], to evaluate differences in scalability for solvers without support for resolving all solutions, as described in Section 4.1. Similarly to ATF, we have written a parser to use PySMT-specific operations where applicable.

We have published the implementation of this evaluation in a repository for further reference.

5.2 Synthetic Tests

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

5.2.1 Experimental setup. We generate a set of search spaces with a varying number of dimensions (between 2 and 5), target Cartesian sizes (with $\{1\times10^4, 2\times10^4, 5\times10^4, 1\times10^5, 2\times10^5, 5\times10^5, 1\times10^6\}$), and number of constraints (between 1 and 6). While these arbitrary parameters result in search spaces that are not as large and do not have as many tunable parameters as the real-world search spaces evaluated on in Section 5.3, the goal of these in total 78 synthetic search spaces is to gain insight into which solver provides good scalability across the variations in these factors.

Given a Cartesian size, a number of dimensions, and a number of constraints, we want to generate a synthetic search space. To prevent an unfair advantage to solvers optimized for a limited number of dominant dimensions, this number of values per dimension v is kept approximately uniform. This is done by first determining the number of values per dimension as $v = s^{\frac{1}{d}}$, where s is the desired Cartesian size and d is the desired number of dimensions. For each of the dimensions, a linear space with v number of elements is instantiated. Given a non-integer value of v, this is rounded to an integer for all but the last dimension, where v is rounded contradictory (e.g. $5.8 \rightarrow 5$, $5.2 \rightarrow 6$) to be closer to the desired Cartesian size.

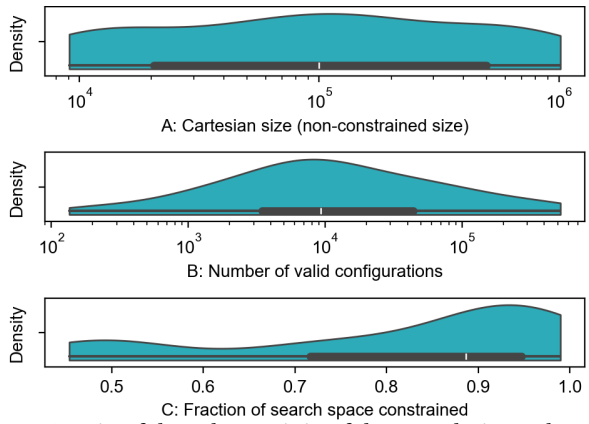

Figure 2: **Density of three characteristics of the 78 synthetic search spaces. Black bottom bar marks the interquartile range and the white line the median.**

A list of constraints involving a variety of operations is generated for each combination of dimensions, which are randomly chosen up to the desired number of constraints.

Figure 2 shows the distribution of the resulting 78 search spaces for three characteristics. Figure 2A shows the actual Cartesian size, representing the total number of possible configurations before constraints are applied, which correspond to the set of target values. Figure 2B depicts the number of valid configurations remaining after constraints are enforced, which follows a bell-shaped curve. The number of valid configurations is on average one order of magnitude below the Cartesian size. Finally, Figure 2C displays the fraction of sparsity of the search space, i.e. the fraction of non-valid configurations relative to the Cartesian size. Though the fraction of constrained configurations is skewed toward higher values, indicating a propensity towards sparsity, a wide range of variations in sparsity is present.

5.2.2 Results. The performance of the evaluated methods on these synthetic search spaces is displayed in various plots in Figure 3, where the colors used correspond to the colors of the methods in the Figure 3C barplot. To determine the impact of the optimizations described in Section 4.3, the *original* method denotes the use of vanilla *python-constraint* before the optimizations, whereas our *optimized* method includes the optimizations of Section 4.3.

Figure 3A shows a clear positive correlation between the number of valid configurations and the execution time across all methods, with a roughly linear trend on the log-log scale, suggesting a power-law relationship. We overlay a log-log linear regression to further investigate the scaling of each method, where a lower slope indicates better scaling towards larger search spaces in the number of valid configurations, and a slope of 1 indicates linear scaling. All methods have a highly significant linear fit with a p-value of 0.000.

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

Figure 3B presents the performance as a continuous probability density curve using a kernel density estimate (KDE). As expected due to their practically linear scaling, ATF and pyATF demonstrate wide variance in performance. Our *optimized* solver consistently achieves the lowest execution times, with several orders of magnitude better performance compared to the other solvers.

Figure 3C summarizes the performance of each solver over all search spaces. It is remarkable that *pyATF* takes considerably longer than the *brute-force* method on these search spaces, which might be due to how optimized the chain-of-trees approach is to highly sparse search spaces. It is also noteworthy that our *optimized* method outperforms the *original* unoptimized implementation of *python-constraint* by several orders of magnitude, demonstrating the advantage of our optimizations. Our *optimized* method achieves a 96x speedup over the *brute-force* method (4.75 seconds versus 455.3 seconds), a 16x speedup over *ATF*, and a 2547x speedup over *pyATF*.

As described in Section 4.1, a traditional solver without support for finding all solutions requires adding the previous solution as a constraint and iterating over the solutions until all solutions have been found. To demonstrate the lack of scalability of such a solver, Figure 4 compares *PySMT* using the Microsoft Z3 solver to the *brute-force* method and our *optimized* method. To make executing this experiment feasible, we reduce the size of the generated synthetic search spaces by one order of magnitude in this experiment. As seen in Figure 4, PySMT performs poorly relative to both brute force and our optimized method. As expected, this difference increases as the number of valid configurations increases, demonstrating the infeasibility of this approach when many valid configurations are present. Despite the reduced search space sizes, PySMT with the Z3 solver still takes nearly a thousand seconds on the largest search spaces, whereas the brute-force solver takes about ten seconds. Our optimized solver vastly outperforms PySMT, taking about as long to solve the largest search spaces as PySMT with the Z3 solver takes to solve the smallest search spaces. In fact, the PySMT solver exhibits superlinear scaling with a slope of 1.090, as opposed to the slope of 0.649 of our optimized method. As it is infeasible to evaluate the large search spaces of the selected real-world applications, PySMT with the Z3 solver will not be included in the remainder of the evaluation.

5.3 Real-world Applications

To evaluate solver performance on the search spaces of real-world applications, we select the three largest search spaces in the Benchmark suite for Auto-Tuners (BAT) [34]. These are *Dedispersion*,

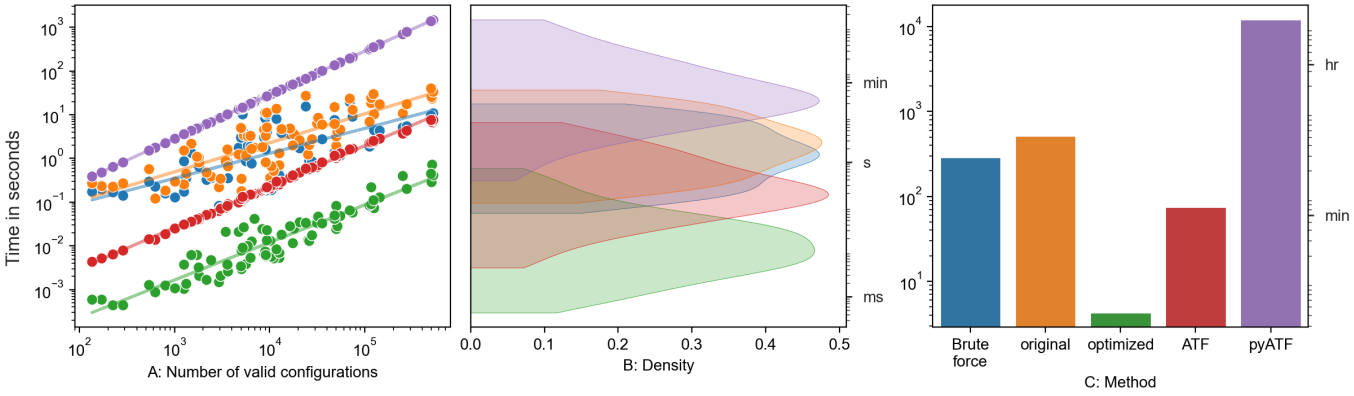

Figure 3: **Search space construction performance on synthetic tests. Lower times are better. Colors correspond to Figure 3C barplot methods. Each plot provides a different view of the same data, with A and B showing the performance on individual search spaces, and C showing the sum of all search spaces.**

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

Table 2: **Overview of the basic characteristics of the real-world search spaces and the mean values for each of the columns.**

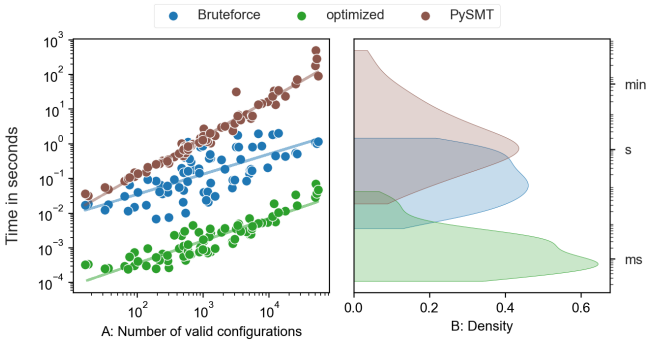

Figure 4: **Search space construction performance of PySMT on synthetic tests.**

Hotspot, and *ExpDist*. In addition, we use the relatively large search spaces of the *MicroHH* computational fluid dynamics kernel [35], as well as the commonly used General Matrix Multiplication kernel *(GEMM)* [24]. To provide reference points for a fair comparison to ATF, the Probabilistic Record Linkage (*PRL*) kernel used in the chain-of-trees evaluation [29] is used as well, resulting in three additional search spaces for a total of eight real-world search spaces.

The characteristics of the real-world search spaces are displayed in Table 2, where the rightmost column denotes the average number of constraint evaluations that are required to brute-force solve a search space. For each combination in the Cartesian product, all constraints need to be evaluated until the combination is rejected or all constraints have been evaluated. Hence, assuming uniform probability among the constraints, the average number of constraint evaluations can be calculated by taking the average of the best case (the first constraint rejects the combination) and worst case (the last constraint rejects the combination), and adding all valid combinations which are never rejected. Given a search space S, let S_i be the set of non-valid combinations, S_v the set of valid combinations, and S_c the set of constraints, the average number of constraint evaluations can be calculated as $\frac{|S_i|+|S_i|\cdot|S_c|}{2}+|S_v|$. Descriptions of each of the kernels and their search spaces are given in Sections 5.3.1 to 5.3.6, before the results are discussed in Section 5.3.7.

5.3.1 Dedispersion. The Dedispersion kernel introduced in [32] and used in [34] is designed to compensate for the time delay experienced by radio waves as they propagate through space. This delay occurs due to the frequency-dependent dispersion of the signal. By applying a specific dispersion measure (DM) and reversing the dispersion effect, the kernel reconstructs the original signal. During the iteration over frequency channels, threads process multiple time samples and dispersion measures in parallel. Comparing the Dedispersion search space to the other search spaces evaluated on in Table 2, the resulting search space is the smallest in Cartesian size, but as it has the highest percentage of valid configurations at nearly 50 %, it is not the smallest in number of valid configurations.

5.3.2 ExpDist. The ExpDist kernel described in [34] is utilized in a localization microscopy application that performs template-free particle fusion by integrating multiple observations into a single super-resolution reconstruction [17]. During the registration process, the ExpDist kernel is repeatedly invoked to evaluate the alignment of two particles. The algorithm exhibits quadratic complexity with respect to the number of localizations per particle, making it highly computationally intensive. The resulting search space is the second-most sparse of the real-world search spaces in Table 2.

5.3.3 Hotspot. Previously discussed in Section 2, the Hotspot kernel of [34] is part of a thermal simulation application to estimate the temperature of a processor by considering its architecture and

simulating power currents. Through an iterative process the kernel solves a set of differential equations. The inputs to the kernel consist of power and initial temperature values, while the output is a grid displaying average temperature values across the chip. It is interesting to note that the Hotspot search space is the largest in number of valid configurations, second-largest in Cartesian size, and has the highest number of values for a single parameter.

5.3.4 MicroHH. The computational fluid dynamics kernel of [35] is used for weather and climate modeling, specifically for the simulation of turbulent flows in the atmospheric boundary layer. In this case, we use the search space resulting from the auto-tunable GPU implementation of the `advec_u` kernel with extended parameter values as specified in the source of [15]. Looking at Table 2, it is notable that the MicroHH search space is the closest to the mean values of all search spaces in the number of parameters, number of constraints, and percentage of configurations. It is also second-closest in constraint size and number of values per parameter, making it perhaps the most average search space in our set of tests.

5.3.5 GEMM. Generalized dense matrix-matrix multiplication is a fundamental operation in the BLAS linear algebra library and widely used across various application domains. Known for its high performance on GPU hardware, GEMM frequently serves as a benchmark in studies of GPU code optimization [20, 25, 30]. In this evaluation, we use the GEMM kernel of CLBlast [24], a tunable OpenCL BLAS library. GEMM is implemented as the multiplication of two matrices (A and B); $C = \alpha A \cdot B + \beta C$, where α and β are constants and C is the output matrix. The dimensions of all three matrices are set to 4096×4096, resulting in a dense search space.

5.3.6 ATF PRL. The Probabilistic Record Linkage (PRL) kernel used in [29] is a parallelized implementation of an algorithm that is commonly used in data mining to identify data records referring to the same real-world entity. In this kernel, the input sizes determine the size of the search space. As shown in Table 2, the brute-force resolution of this search space with input sizes $8x8$ requires 1.8119×10^{10} constraint evaluations on average, which took ~27 hours to execute. As an input size of $16x16$ would require 4.639×10^{12} constraint evaluations on average, it is not feasible to brute force beyond the $8x8$ input size. Because the brute-forced solution is used for validation and serves as a reference point in the performance comparisons, we use the search spaces resulting from the ATF PRL kernel with input sizes $2x2$, $4x4$, and $8x8$. It is notable that while the $8x8$ search space results in the largest Cartesian size of the set, the ATF PRL search spaces are very sparse.

5.3.7 Results. Figure 5 presents the search space construction performance across the eight real-world benchmarks for the five different constraint solver methods as in Section 5.2.

Figure 5A and Figure 5B illustrate the relationship between search space size and solver performance, with a log-log linear regression overlayed where significant (p-value ≤ 0.05), as in Section 5.2.2. In general, larger constrained search spaces (A) and Cartesian sizes (B) result in increased search times, as previously observed in Figure 3. For the *ATF*, *original*, and *brute-force* methods the significant scaling trend is along the Cartesian size of Figure 5B, whereas for *pyATF* and our *optimized* method this is on the number of valid configurations in Figure 5A. Our *optimized* method achieves the lowest execution times across all search space sizes, demonstrating its efficiency, and is the only solver that consistently outperforms the other methods.

Figure 5C visualizes the distribution of execution times, providing an indication of the average performance and variability. It is interesting to observe that while the *original* python-constraint method is one order of magnitude faster than the *brute-force* method, both methods have very similar distributions, as seen before in Figure 3B. A clear trend emerges from this plot, where our *optimized* method has the least variability and is the only solver constructing the search spaces in the sub-second domain.

In Figure 5D, the relation between how constrained a search space is and solver performance is displayed. *ATF* and *pyATF* performance appears to be strongly influenced by the sparsity, as for fraction > 0.9 *ATF* performance is substantially better than the *original* solver, in contrast to ≤ 0.9, where at fraction $\simeq 0.5$ even the unoptimized *original* python-constraint outperforms *ATF*.

The number of tunable parameters displayed in Figure 5E do not appear to have as much of an impact on performance as the other plots discussed. Nevertheless, Figure 5E is useful to discern the individual search spaces based on the number of parameters described in Table 2. For instance, it can be noted that the performance difference between our optimized method and all other methods appears to be relatively stable, even for the ATF PRL search spaces, as can be discerned by the number of tunable parameters, where the three ATF search spaces have 20 tunable parameters.

Finally, Figure 5F summarizes the total time taken by each solver. The brute-force approach is the least performant, taking nearly a full day to resolve the eight search spaces. Although the *original* python-constraint solver is faster than brute force, our *optimized* solver achieves a ~2643x speedup over it, demonstrating the efficiency of our optimizations. While ATF and pyATF achieve intermediate performance levels, it must be noted that pyATF only outperforms *brute force* and *original* because it does so on the two largest PRL search spaces, which have a disproportionate effect on the summed time - on all other search spaces pyATF is outperformed by both methods, and ATF is not consistently better than *original* either. The optimized solver consistently and considerably outperforms all others: our *optimized* method achieves a ~20611x overall speedup over the brute-force method (3.16 seconds versus 65230.47 seconds), ~44x over ATF, and ~891x over pyATF.

5.4 Overall impact in practice

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

We auto-tune the *hotspot* kernel described in Section 5.3.3 using the three Python-based solvers with a 30-minute time budget as an illustrative example. To avoid influence by a specific optimization algorithm we use random sampling, and each run is repeated 10 times. Figure 6 shows the best-performaning configuration found so far during the tuning process, where higher is better. The time passed before any configuration is found is spent constructing the search space, which takes about eight minutes for brute-force and takes over twenty minutes for pyATF, while our optimized method is able to start to tune configurations almost immediately.

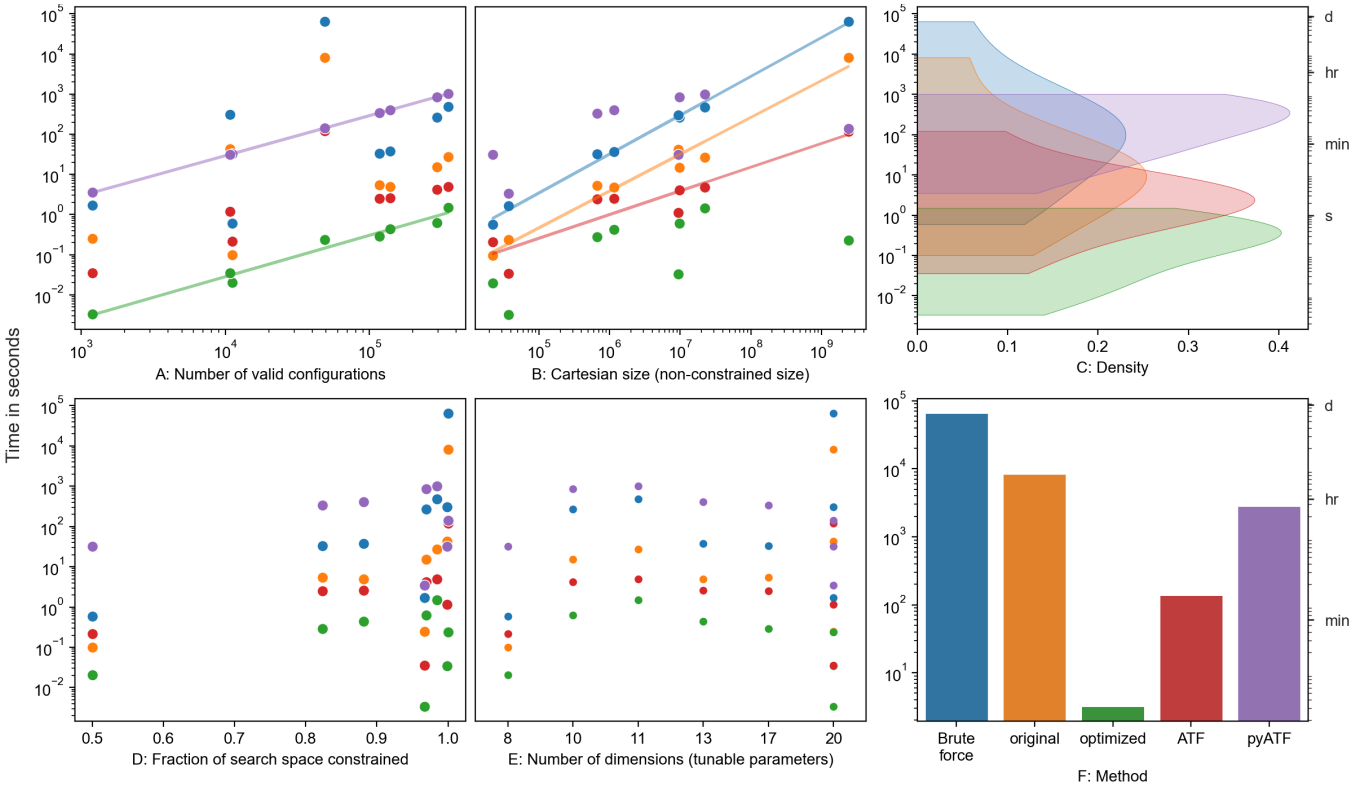

Figure 5: **Search space construction performance on real-world tests. Lower times are better. Colors correspond to Figure 5F barplot methods. Each plot provides a different view of the same data; plots A-E show individual performance relative to a search space charactistic, and F shows the sum of all search spaces.**

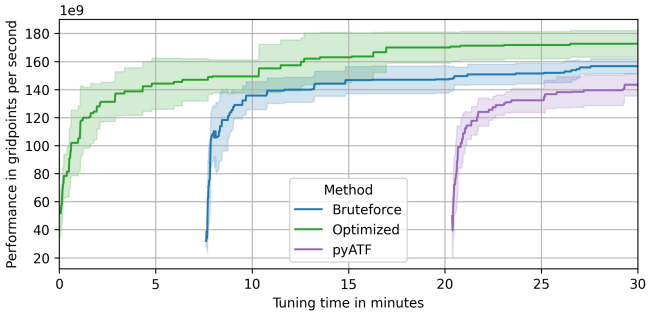

Figure 6: **Best configuration performance found over a 30-minute auto-tuning of the *hotspot* kernel using various search space construction methods.**

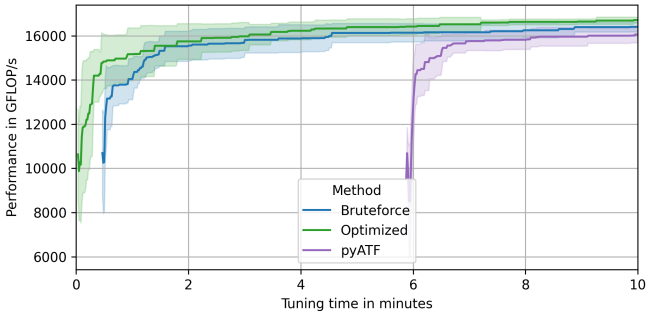

Figure 7: **Best configuration performance found over a 10-minute auto-tuning of the *GEMM* kernel using various search space construction methods.**

To affirm these findings we repeat this experiment on the *GEMM* kernel, adjusting the time budget by the ratio between the number of valid configurations of GEMM and *hotspot* seen in Table 2 to 10 minutes. While brute force fares substantially better, as expected due to the smaller and denser search space, the results are otherwise very similar to that of the previous experiment. Most importantly, both examples confirm that the search space construction method has a substantial impact on the quality of the overall best configuration found within the time budget.

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

6 Conclusions

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

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

This breakthrough allows researchers and developers to more effectively harness the performance potential of modern hardware and provides an efficient generic solver for similar problem domains.

Availability: The methods presented in this work are available as user-friendly software packages, enabling straightforward adoption by the auto-tuning community and related fields. They can be installed with `pip install python-constraint2` and `pip install kernel-tuner`. Both python-constraint and Kernel Tuner are open-source software welcoming contributions. For more information, visit the Kernel Tuner[1] and python-constraint[2] repositories.

References

[1] Z. Alomari, O. E. Halimi, K. Sivaprasad, and C. Pandit. 2015. Comparative Studies of Six Programming Languages.

[2] J. Ansel, S. Kamil, et al. 2014. Opentuner: An extensible framework for program autotuning. In *23rd international conference on Parallel architectures and compilation*.

[3] M. Ateeq, H. Habib, A. Umer, and M. U. Rehman. 2014. C++ or Python? Which One to Begin with: A Learner's Perspective. In *Int. Conf. Teach. Learn. Comput. Eng.*

[4] H. Bal, D. Epema, C. de Laat, R. van Nieuwpoort, J. Romein, F. Seinstra, C. Snoek, and H. Wijshoff. 2016. A Medium-Scale Distributed System for Computer Science Research: Infrastructure for the Long Term. *Computer* (2016).

[5] P. Balaprakash, J. Dongarra, T. Gamblin, M. Hall, J. K. Hollingsworth, B. Norris, and R. Vuduc. 2018. Autotuning in High-Performance Computing Applications. *Proc. IEEE* (2018).

[6] C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli. 2008. Satisfiability Modulo Theories. In *Handbook of Satisfiability*.

[7] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn, and K. Smith. 2011. Cython: The Best of Both Worlds. *Comput. Sci. Eng.* (2011).

[8] A. Biere, A. Biere, M. Heule, H. van Maaren, and T. Walsh. 2009. *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*.

[9] N. Bjørner, L. de Moura, L. Nachmanson, and C. M. Wintersteiger. 2019. Programming Z3. In *Engineering Trustworthy Software Systems*.

[10] S. C. Brailsford, C. N. Potts, and B. M. Smith. 1999. Constraint Satisfaction Problems: Algorithms and Applications. In *Eur. J. Oper. Res.*

[11] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. 2009. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Int. Symp. Workload Charact. IISWC*.

[12] L. De Moura and N. Bjørner. 2008. Z3: An Efficient SMT Solver. In *Int. Conf. Tools Algorithms Constr. Anal. Syst.*

[13] T. L. Falch and A. C. Elster. 2015. Machine Learning Based Auto-Tuning for Enhanced OpenCL Performance Portability. In *2015 IEEE Int. Parallel Distrib. Process. Symp. Workshop*.

[14] M. Frigo and S. G. Johnson. 1998. FFTW: An adaptive software architecture for the FFT. In *Acoust. Speech Signal Process.*

[15] S. Heldens and B. van Werkhoven. 2023. Kernel Launcher: C++ Library for Optimal-Performance Portable CUDA Applications.

[16] E. O. Hellsten, A. Souza, J. Lenfers, R. Lacouture, O. Hsu, A. Ejjeh, F. Kjolstad, M. Steuwer, K. Olukotun, and L. Nardi. 2024. BaCO: A Fast and Portable Bayesian Compiler Optimization Framework. In *Proc. 28th ACM Int. Conf. Archit. Support Program. Lang. Oper. Syst. Vol. 4*.

[17] H. Heydarian, F. Schueder, M. T. Strauss, B. van Werkhoven, M. Fazel, K. A. Lidke, R. Jungmann, S. Stallinga, and B. Rieger. 2018. Template-Free 2D Particle Fusion in Localization Microscopy. *Nat. Methods* (2018).

[18] P. Hijma, S. Heldens, A. Sclocco, B. Van Werkhoven, and H. E. Bal. 2023. Optimization Techniques for GPU Programming. *ACM Comput. Surv.* (2023).

[19] J. Lawson, M. Goli, D. McBain, D. Soutar, and L. Sugy. 2019. Cross-Platform Performance Portability Using Highly Parametrized SYCL Kernels. *ArXiv* (2019).

[20] Y. Li, J. Dongarra, and S. Tomov. 2009. A Note on Auto-Tuning GEMM for GPUs. In *Computational Science – ICCS*.

[21] Y. Liu, W. M. Sid-Lakhdar, O. Marques, X. Zhu, C. Meng, J. W. Demmel, and X. S. Li. 2021. GPTune: Multitask Learning for Autotuning Exascale Applications. In *Proc. 26th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*

[22] G. LLC. 2015. Ortools: Google OR-Tools Python Libraries and Modules.

[23] G. Niemeyer. 2005. Python-Constraint: a Module Implementing Support for Handling CSPs over Finite Domain.

[24] C. Nugteren. 2018. CLBlast: A Tuned OpenCL BLAS Library. In *Proc. Int. Workshop OpenCL*. Article 5.

[25] C. Nugteren and V. Codreanu. 2015. CLTune: A generic auto-tuner for OpenCL kernels. In *9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip*.

[26] F. Petrovič and J. Filipovič. 2023. Kernel Tuning Toolkit. *SoftwareX* (2023).

[27] F. Petrovič, D. Střelák, J. Hozzová, J. Ol'ha, et al. 2020. A Benchmark Set of Highly-Efficient CUDA and OpenCL Kernels and Its Dynamic Autotuning with Kernel Tuning Toolkit. *Future Gener. Comput. Syst.* (2020).

[28] A. Rasch and S. Gorlatch. 2018. ATF: A Generic Directive-based Auto-tuning Framework. *Concurr. Comput. Pract. Exp.* (2018).

[29] A. Rasch, R. Schulze, M. Steuwer, and S. Gorlatch. 2021. Efficient Auto-Tuning of Parallel Programs with Interdependent Tuning Parameters via Auto-Tuning Framework (ATF). *ACM Trans Arch. Code Optim*, Article 1 (2021).

[30] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S.-Z. Ueng, J. A. Stratton, and W.-mei W. Hwu. 2008. Program Optimization Space Pruning for a Multithreaded Gpu. In *Proc. 6th Annu. IEEEACM Int. Symp. Code Gener. Optim.*

[31] R. Schulze, S. Gorlatch, and A. Rasch. 2025. pyATF: Constraint-based Auto-Tuning in Python. In *Proc. 34th ACM SIGPLAN Int. Conf. Compil. Constr.*

[32] A. Sclocco, H. E. Bal, J. Hessels, J. van Leeuwen, and R. V. van Nieuwpoort. 2014. Auto-Tuning Dedispersion for Many-Core Accelerators. In *2014 IEEE 28th Int. Parallel Distrib. Process. Symp.*

[33] P. Team. 2022. PySMT: A Solver-Agnostic Library for SMT Formulae Manipulation and Solving.

[34] J. O. Tørring, B. van Werkhoven, F. Petrovč, F.-J. Willemsen, J. Filipovič, and A. C. Elster. 2023. Towards a Benchmarking Suite for Kernel Tuners. In *IEEE Int. Parallel Distrib. Process. Symp. Workshop IPDPSW*.

[35] C. C. Van Heerwaarden, B. J. Van Stratum, T. Heus, J. A. Gibbs, E. Fedorovich, and J. P. Mellado. 2017. MicroHH 1.0: A Computational Fluid Dynamics Code for Direct Numerical Simulation and Large-Eddy Simulation of Atmospheric Boundary Layer Flows. *Geosci. Model Dev.* (2017).

[36] B. van Werkhoven. 2019. Kernel Tuner: A Search-Optimizing GPU Code Auto-Tuner. *Future Gener. Comput. Syst.* (2019).

[37] B. van Werkhoven, J. Maassen, H. E. Bal, and F. J. Seinstra. 2014. Optimizing Convolution Operations on GPUs Using Adaptive Tiling. *Future Gener. Comput. Syst.* (2014).

[38] R. C. Whaley, A. Petitet, and J. J. Dongarra. 2001. Automated Empirical Optimizations of Software and the ATLAS Project. *Parallel Comput.* (2001).

[39] F.-J. Willemsen, R. van Nieuwpoort, and B. van Werkhoven. 2021. Bayesian Optimization for Auto-Tuning GPU Kernels. In *Int. Workshop Perform. Model. Benchmarking Simul. High Perform. Comput. Syst. PMBS*.

[40] X. Wu, P. Balaprakash, M. Kruse, J. Koo, B. Videau, P. Hovland, V. Taylor, B. Geltz, S. Jana, and M. Hall. 2024. Ytopt: Autotuning Scientific Applications for Energy Efficiency at Large Scales. *Concurrency and Computation* (2024).

[41] F. Zehra, M. Javed, D. Khan, and M. Pasha. 2020. Comparative Analysis of C++ and Python in Terms of Memory and Time.

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

[2] https://github.com/python-constraint/python-constraint