Efficient Construction of Large Search Spaces for GPU autotuning

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

Floris-Jan Willemsen
Leiden University
Leiden, the Netherlands
Netherlands eScience Center
Amsterdam, the Netherlands

Rob van Nieuwpoort
Leiden University
Leiden, the Netherlands

Ben van Werkhoven
Leiden University
Leiden, the Netherlands
Netherlands eScience Center
Amsterdam, the Netherlands

Abstract

Graphics Processing Units (GPUs) have become indispensable as a computing resource due to their exceptional computational performance for data- and compute-intensive tasks. Auto-tuning is widely used to optimize the performance, accuracy, and energy efficiency of GPU programs by selecting the best kernel configurations from a vast search space. However, as GPU architectures and applications become more complex, the demands on auto-tuners have increased significantly. The explosion of possible kernel configurations — often exceeding millions — has made the construction of these search spaces a bottleneck in the tuning process.

This paper addresses this challenge by leveraging Constraint Satisfaction Problem (CSP) solvers to efficiently construct and represent GPU kernel search spaces. We introduce several optimizations to enhance the solver's efficiency and integrate our approach into the open-source auto-tuning framework Kernel Tuner. Our methods significantly reduce the overhead of search space construction while maintaining flexibility and scalability. Experimental results show that our approach improves search space construction performance by several orders of magnitude, making auto-tuning feasible for large and complex GPU search spaces.

CCS Concepts

• **Computing methodologies** → *Discrete space search*; Parallel programming languages.

Keywords

Search spaces, Constraint satisfaction, Constraint solving, Discrete space, Autotuning, GPU applications, Kernel Tuner, Optimization problems

ACM Reference Format:
Floris-Jan Willemsen, Rob van Nieuwpoort, and Ben van Werkhoven. 2025. Efficient Construction of Large Search Spaces for GPU autotuning: ICS 2025 Submission **#NaN** – Confidential Draft – Do NOT Distribute!!. In *Proceedings of The 39th ACM International Conference on Supercomputing ('ICS 2025')*. ACM, New York, NY, USA, 12 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Graphics Processing Units (GPUs) have revolutionized the computing landscape in the past decade, providing previously unattainable computational performance for compute-intensive tasks such as artificial intelligence and climate model simulation [20, 26]. Nine out of the top ten supercomputers in the Top 500 use GPUs as the main source of compute power, and systems with accelerators account for 82.6% of the combined top500 RMax performance [1]. GPUs excel in terms of compute performance and energy efficiency for tasks that involve large data sets and dense computation, making them increasingly vital in various scientific domains [51]. In the past decade, GPUs have become increasingly complex computing devices with larger register files, more specialized cores, and larger and more complex streaming multiprocessors (SMs), while also dramatically increasing the number of SMs per chip [24]. In addition, energy efficiency and accuracy play an increasingly important role in the auto-tuning of GPU applications [21, 46].

GPU programming models, such as CUDA, HIP, and OpenCL, allow developers to create highly parallel functions, called *kernels* that run on the GPU. GPU programmers are confronted with a myriad of implementation choices and optimization techniques related to thread organization, memory usage, and computation strategies to achieve optimal compute performance [24]. Many different design choices have a substantial and hard-to-predict impact on the performance, energy efficiency, and accuracy of GPU kernels, as the optimal kernel configuration depends on a complex interplay of hardware, device software, and the program itself. This optimization problem leads to an overwhelming number of code variants if done manually, spurring the creation of frameworks that facilitate automatic performance tuning, or *auto-tuning*, to automatically tune GPU kernels and related software [1, 16, 36, 39, 50]. As a consequence of the widespread adoption of GPUs for computation, increased complexity of GPUs, and improvements in auto-tuning, the number of parameters to be tuned is increasing, as well as the range of values per parameter. This is reflected in auto-tuned GPU applications, with the number of valid kernel code variants, or *configurations*, per search space at millions and approaching billions in recent work [21, 22, 25].

The dramatically increased search space size creates new challenges for auto-tuning GPU applications. Auto-tuning frameworks and their users employ constraints on the tunable parameters to avoid attempts to tune invalid or non-sensical configurations, for example, an invalid product of thread block sizes. As such, creating and representing the search space itself, with billions of possible combinations to resolve in a high-dimensional, discontinuous space quickly becomes a bottleneck at the start of the tuning process [41]. For several real-world tunable applications, the search space construction time can take several minutes or even hours (measured per the experimental setup described in Section 4). This is time that could have been spent on tuning, but is instead lost to the overhead

[1] https://top500.org/lists/top500/2024/11/ (Accessed February 2025)

'ICS 2025', June 9–11, 2025, Salt Lake City, USA
2025. ACM ISBN 000-0-0000-0000-0/00/00
https://doi.org/XXXXXXX.XXXXXXX

of constructing the search space; for example, the brute force search space construction of the Hotspot kernel used in Section 4 takes over 8 minutes, during which the GPU is not used and the user receives no tangible result or feedback.

To address this issue, this publication introduces the use of Constraint Satisfaction Problem (CSP) solvers and various optimizations for constructing auto-tuning search spaces, dramatically improving over the state-of-the-art auto-tuners for fast search space construction. In particular, we examine and evaluate various solver techniques and implementations to decide which approach is best suited for auto-tuning, and greatly optimize the best approach by creating C-extensions, efficiently parsing inputs, extending and optimizing built-in constraints, and efficiently representing the resulting search space. Our contributions have been implemented in python-constraint [2] and Kernel Tuner [50, 55] [3], both available as open-source packages.

The remainder of this publication is structured as follows. Section 2 discusses related work. Section 3 describes the selection (Section 3.2), optimization (Section 3.3), and implementation (Section 3.4) of search space solvers for constructing auto-tuning search spaces. In Section 4, we evaluate the efficiency and scalability of our optimized CSP-based approach against various other state-of-the-art solutions on a wide variety of synthetic and real-world search spaces. Section 5 concludes this work.

2 Related Work

In this section, related works are discussed to provide context on the developments and current state of auto-tuning, gradually focusing on search spaces. There are many different automated approaches to improving the performance of software that are collectively referred to as auto-tuning. For an early survey of different uses of auto-tuning in high-performance computing, see Balaprakash et al.. As described in this survey, at the heart of every auto-tuning approach is a *search space* of *code variants* that affect code organization, data structures, high-level algorithms, or low-level implementation details, while remaining functionally equivalent to some original implementation [4].

There are several different axes along which auto-tuning approaches can be compared. For example, an auto-tuner can be application-specific, e.g., FFTW [17], ATLAS [54], or *generic*, meaning it can be used to optimize any application. Auto-tuners may use different approaches to score code variants, relying either on some performance model [43] or on empirical measurements using the targeted hardware. Some auto-tuners optimize applications at compile-time towards a specific hardware architecture, while others aim to optimize application performance at runtime, creating a distinction between auto-tuning during development ("offline") or execution ("online"). Some auto-tuning frameworks focus on minimizing the execution time of whole applications [1, 28, 33, 56], whereas others focus on the optimization of individual functions [16, 39, 50]. A comprehensive overview of the field of auto-tuning research is beyond the scope of this work. As such, we limit our discussion to works that auto-tune individual functions for GPUs (also known as kernels) at compile-time.

An important distinction in auto-tuning is how code variants are created. There are generally two approaches, known as *compiler-based* or *software-level* auto-tuning. In compiler-based auto-tuning, the user implements only a single version of the code and a compiler is responsible for generating different, functionally equivalent, code variants that exhibit different performance when executed. Software-level auto-tuning on the other hand generally leaves the responsibility of specifying different code variants with the programmer, using for example metaprogramming approaches, such as code generators, macros, and templates. We discuss related work from both approaches in the next Sections 2.1 and 2.2.

2.1 Compiler-based auto-tuning

Several compiler-based auto-tuning approaches for GPU kernels have been presented in the past two decades.

Orio [19] is a framework that transforms annotated kernels to target languages, incorporating an auto-tuning phase to select optimal compiler optimization parameters. BOAST [53] is a metaprogramming framework built on top of Orio that targets high-performance computing by simplifying application optimization through a high-level interface language. BOAST selects compiler optimizations based on user-specified kernels and options. The Adaptive Sampling Kit (ASK) [13] employs active learning to efficiently sample large search spaces, providing various methods for sampling. Coding Ants [37] uses ant colony optimization for auto-tuning. The framework is built as an extension of the Polyhedral Parallel Code Generator (PPCG [52]) for generating CUDA code from C code. Ashouri et al. wrote a comprehensive survey on machine-learning methods for compiler-based auto-tuning.

Compiler-based auto-tuning generates and tunes code variants as part of the compilation process. As there is no direct user input on the code variants and the order in which optimization passes are applied matters, the search space construction process of compiler-based auto-tuning is substantially different from the software-level auto-tuners we will focus on.

2.2 Software-level auto-tuning

Over the last decade, several software-level auto-tuning frameworks for GPU kernels have been introduced.

AUMA [14] utilizes neural network models for auto-tuning the performance of OpenCL kernels on Intel CPUs, as well as GPUs from Nvidia and AMD, intending to enable performance portability across different hardware architectures. It is evaluated using three benchmark kernels: *convolution*, *raycasting*, and *stereo*. Respectively, their search spaces consist of 131072, 655360, and 2359296 configurations.

Dao and Lee [11] present an auto-tuner for tuning the workgroup size of OpenCL kernels with an extensive evaluation of 54 OpenCL kernels on 4 different GPUs. Given that they only tune the workgroup sizes in at most two dimensions, the resulting search spaces are relatively small.

CLTune [36] is another open-source framework for auto-tuning for OpenCL kernels. It was the first framework to employ parameter insertion using preprocessor macros and supports validation by a reference kernel. CLTune implements several optimization

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

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

algorithms to accelerate the auto-tuning process, including simulated annealing and particle swarm optimization. The CLTune framework is evaluated on two kernels [36], a 2D convolution and matrix multiplication. The convolution kernel has a Cartesian size of 12288 parameter combinations, of which 3424 are valid configurations (28%). The matrix multiplication kernel has a Cartesian size of 2654208, of which 995328 are valid configurations (37.5%). Source code inspection reveals CLTune uses brute force search space construction by recursively iterating over all permutations of the user-defined parameters.

OpenTuner [1], an open-source framework introduced in 2014, supports multiple languages but does not specifically target GPUs and requires manual host code implementation for each kernel. OpenTuner optimizes the search space using different techniques simultaneously. Source code inspection confirmed OpenTuner uses brute force search space construction by applying constraints in mapping over all permutations of the user-defined parameters. OpenTuner is evaluated on applications [1] such as Halide, High-Performance Linpack, and PetaBricks. While the true number of configurations is not specified, the Cartesian sizes listed range from $10^{6.5}$ to 10^{6328}.

Kernel Tuning Toolkit (KTT) [16] is an open-source auto-tuning framework that supports both compile-time and online auto-tuning, where code variants are tested while the application is running in production. KTT supports auto-tuning of Vulkan, CUDA, and OpenCL kernels. Filipovič et al. [15] extended KTT with a machine-learning approach that incorporates performance counter data collected by a profiler to accelerate the auto-tuning process. KTT constructs the search space by using a tree-based resolution, which can be resolved in parallel when creating independent subspaces (called groups) that do not share constraints, which are used when tuning composite kernels and must be labeled as separate groups by the user. By default a single group is used, resulting in sequential recursive resolution. In the evaluation section [16], three kernels are used: a 2D and 3D Coulomb summation, and a reduction kernel. Both Coulomb summations use the same parameters, resulting in a Cartesian size of 16128 and 14784 valid configurations (91.7%), and the reduction kernel has a Cartesian size of 10080 parameter combinations and 2640 valid configurations (26.2%). It must be noted that the parameters used in the publication differ from those in the source code [4]; the source definition with widest parameter values is used here as the constraints are not specified in the publication.

The most closely related work is on Auto-Tuning Framework (ATF) [39, 41], as this is the state-of-the-art in search space construction. ATF can do efficient search space construction for large optimization spaces with interdependent parameters using chain-of-trees [41]. ATF supports auto-tuning of GPU kernels through a domain-specific language and is available as an open-source library in both C++ and Python implementations (referred to as *ATF* and *pyATF* respectively). ATF [39] evaluates on the *XgemmDirect* kernel of [35] with four different input sizes, resulting in four search spaces. The parameters used in the publication have substantially more values than those in the source code at the reported version 0.11.0, but do not report enough detail to determine the search space sizes, which have therefore been omitted.

The Bayesian Compiler Optimization framework (BaCO) is an auto-tuning framework for GPU, CPU, and FPGA applications. It supports a wide variety of parameter types and introduces the concept of "hidden" constraints to auto-tuning, which are discovered during optimization. For search space construction, it uses the chain-of-trees approach introduced in [39] and improves upon the sampling approaches introduced by ATF. BaCO is evaluated on a total of fifteen search spaces from three real-world applications: *TACO*, *RISE & ELEVATE*, and *HPVM2FPGA*. The average Cartesian size for these search spaces is 208006300000, 16821461143, and 285085, respectively. The average number of configurations and the percentage of the Cartesian size are 1961000 (21%), 23471314 (9.7%), and 285085 (100%), respectively.

Kernel Tuner [50], an open-source Python-based software auto-tuner for GPU applications, also belongs to the category of software-level auto-tuning frameworks. Kernel Tuner supports mainstream GPU programming languages such as OpenCL, HIP [30], and CUDA, as well as OpenMP and OpenACC in both C and Fortran. Kernel Tuner is also capable of optimizing GPU kernels for energy efficiency, accuracy, and other custom objectives besides minimizing kernel execution time [46], and provides a wide variety of optimization algorithms [45, 55]. Kernel Launcher [21] is a library that builds on top of Kernel Tuner to facilitate the integration of tuned kernels in C++ applications. The search spaces used in these publications illustrate the general trend of larger search spaces; where in 2019 the number of valid configurations was in the order of thousands, in the tens of thousands in 2022a, 2022b, 2021, and has gotten to millions in 2023. Kernel Tuner has so far used brute force search space construction, which suffices for the original size of auto-tuning problems, but can take a substantial amount of time with the large search spaces currently encountered. As such, in this work, we aim to provide efficient construction of large search spaces for auto-tuning and implement this in Kernel Tuner.

3 Implementation

This section discusses the implementation details of our novel method for efficiently constructing large search spaces for GPU auto-tuning. We first provide the context of this implementation within the Kernel Tuner framework Section 3.1, followed by an examination of various constraint-solving techniques and implementations and discuss which is best suited to auto-tuning Section 3.2. Next, the resulting best approach is further optimized as detailed in Section 3.3, and finally the representation of the resulting Searchspace in Kernel Tuner Section 3.4.

3.1 Implementation context

Kernel Tuner acts as an external tool for developers to benchmark and optimize GPU kernels in isolation, which can be used with applications in any host programming language. The modular software architecture of Kernel Tuner is shown in Figure 1. Users of Kernel Tuner create a small Python script that points to the GPU code and describes both the tunable parameters and any constraints (or *restrictions*) to filter out invalid combinations of parameter values.

[4] https://github.com/HiPerCoRe/KTT/blob/master/Examples/CoulombSum2d/CoulombSum2d.cpp, https://github.com/HiPerCoRe/KTT/blob/master/Examples/Reduction/Reduction.cpp

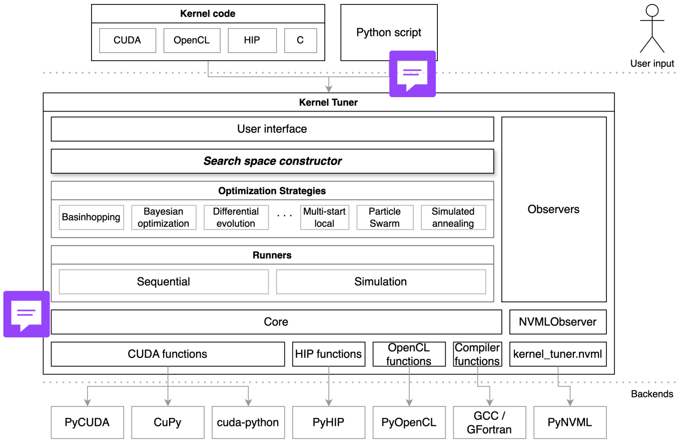

Figure 1: Kernel Tuner software architecture.

There are also many optional settings that users can specify, including derived metrics to be computed, the optimization objective to use, which optimization algorithm to use, and hyperparameters to the optimization algorithm of choice, to name a few.

Search space construction is the first step towards auto-tuning any function or application, preceding the search through the myriad of configurations or code variants. This is apparent from the Kernel Tuner architecture shown in Figure 1, where the modular structure of Kernel Tuner is seen, with processing flowing generally from the top (user input), through the search space construction, to the strategies (optimization algorithms) that determine the next configuration to evaluate, to the runners that prepare the evaluation, and the backends that compile and execute the kernel. After execution, the process flows back up in the diagram, either to the strategies to determine the next configuration based on the new result and repeat the process, or by reporting the tuning results to the user.

In this work, we will focus on the *search space* part of Figure 1, as with possibly millions of code variants to enumerate in a high-dimensional space, where user-defined constraints (also commonly referred to as restrictions) cut out parts of the space that are considered invalid, constructing the search space can become a bottleneck at the start of the tuning process. In GPU auto-tuning, the Cartesian product of the tunable parameters, that is, the collection of all possible combinations of all parameter values, tends to contain many configurations that are not valid. For example, because the product of the thread block sizes can not be larger than some hardware limitation, or because some combination of parameter values would lead to incorrect results in the kernel. To filter out such configurations, the user can specify constraints on certain combinations of tunable parameter values.

Initially, Kernel Tuner simply used a brute-force solution: generate all the combinations of parameter values and filter out combinations based on the user-specified constraint, leaving only valid configurations. This is reasonable for small search spaces but becomes increasingly time-consuming as the search space size and number of constraints increase. Hence, to improve performance on large search spaces, the search space must be constructed more efficiently.

One approach could be to resolve the constraints dynamically, either by only checking the constraints of combinations suggested by the strategy before executing kernel configurations, or by resolving the search space in parallel while executing. However, these dynamic approaches pose problems. If we take the approach of only checking combinations suggested by the strategy, in tuning problems with sparse search spaces (where the vast majority of combinations of parameter values are not valid configurations), a substantial amount of time would be spent on finding any valid configuration as for each combination the strategy needs to be consulted and the combination checked against the constraints. This problem is exacerbated by the fact that during the tuning process, the number of valid non-executed configurations decreases. To illustrate this, in the case of a random search on a search space of 100 valid configurations in a search space where 99% is invalid (10000 combinations) and configurations are taken from the search space once executed, the expected number of attempts required to find a valid first configuration is 100. However, the expected number of attempts required to find the fiftyfirst valid configuration is 199, and the sum of expected attempts required from the first to the fiftyfirst valid configuration is over 7000 - over two-thirds of the total Cartesian size of the space. While this can be mitigated by keeping track of all attempted combinations, this by itself produces a large memory footprint. In addition, dynamic resolution can skew initial sampling and strategies, as the knowledge of the constraints is not fully incorporated in the search.

An example of a dynamic approach to search space resolution in auto-tuning is the chain-of-tree approach used by ATF [39] as described in Section 2. While this is efficient for search spaces where the vast majority of possible combinations are invalid, individual constraints may only use a small subset of all the parameters to achieve this efficiency. In addition, 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, are not resolved with the dynamic approach. Furthermore, randomized sampling is inherently biased to the sparser parts of the tree, although this has been addressed by BaCO [22]. Moreover, selecting neighbors of configurations as extensively used by various optimization algorithms in Kernel Tuner is potentially expensive.

Instead, we aim to 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 search strategies.

3.2 Constraint Solvers in Auto-tuning

In general, this type of problem, where parameter values and constraints are resolved to valid combinations, can be encoded as a Boolean Satisfiability Problem (SAT) [8], Satisfiability Modulo Theories (SMT) [6], or Constraint-Satisfaction Problem (CSP) [10]. At the time of writing, several frameworks are readily available as Python packages that have implemented solvers for these types of problems: *CSP-solver* [32], *Google ORTOOLS* [29], *PicoSAT* [44], *CPMpy* [18], *PyChoco* [38], *SATisPy* [31], *PySMT* [48], *python-constraint*

[34]. Nevertheless, not all of these frameworks can be applied to auto-tuning; they have their limitations in how expressive and efficient they are. In general, the SAT, SMT, and CSP problem types mentioned serve different purposes, owing to their different origins. SAT solvers are generally most efficient, at the cost of expressivity, as they are highly optimized for propositional logic problems. SMT solvers allow for non-integer finite domains such as floating-point numbers, strings, and lists, and as such are more expressive yet not as highly optimized compared to SAT solvers. CSP solvers offer high-level abstractions suitable for modeling complex constraints, providing special types such as "all-different" [47], which are otherwise difficult to efficiently express, making them ideal for complex combinatorial constraints.

In auto-tuning, the constraint problem consists of a set of named parameters, each parameter having a finite list of possible values, usually numeric but also strings or other types. As such, SMT and CSP solvers are the best fit in this case, leaving *CSP-solver*, *PyChoco*, *PySMT*, and *python-constraint*. In addition, there are practical considerations when it comes to choosing a solver; *PyChoco* is in beta and requires building from source, and *PySMT* requires manual steps to install actual solvers, making it cumbersome to deploy as a dependency within a framework. Various solvers, such as *CSP-solver* and *PySMT*, aim to find any solution, rather than all solutions, as required in the case of auto-tuning. 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 can be expected with auto-tuning problems, this hampers performance substantially.

Hence, we focus on *python-constraint*, as this is a CSP-based Python package with built-in support to find all solutions. Initially developed by Gustavo Niemeyer and successively maintained by Sébastien Celles, the *python-constraint* package was first released in 2017. The more than 22000 weekly downloads[5] at the time of writing indicate that has a substantial user base.

3.3 Efficiency Optimizations

To further enhance the performance of *python-constraint*, both in general and for auto-tuning search spaces. Specifically, we implemented several key improvements.

First, we employ 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 tested sequentially, and constraints are verified as black-box functions. If a constraint is violated, the algorithm backtracks by restoring previous states from a queue until all possibilities are explored. By systematically iterating through variable assignments while leveraging heuristics, the algorithm efficiently constructs all valid solutions while minimizing unnecessary exploration.

In addition, we Cythonized the codebase by transpiling from Python to C-code using Cython [7], which is then compiled into Python-importable C-extensions. This process included adding type hints where possible to aid in compilation. Binaries are precompiled for the most common operating systems (Linux, macOS, and Windows) on the supported Python versions (3.9 through 3.13 as of this writing), reducing package installation time and the dependencies required to do so.

Additionally, we expanded and improved built-in constraints to optimize operations that are commonly used. Commonly used operations allow for increased efficiency over generic functions by applying knowledge of the operation. For example, given a constraint where $p \cdot q > 0$, if we have a minimum product constraint specified, we know to ignore all cases where $p \leq 0 \lor q \leq 0$. Hence we added *MaxProduct* and *MinProduct* constraints as they are commonly used in auto-tuning constraints (e.g. a maximum product of block sizes due to hardware limits). We also rewrote and added preprocessing steps to the *Function*, *MaxSum* and *MaxProduct* constraints. Given that the *Function* constraint is both a common occurrence as it is used wherever the more specific constraints do not apply and a generally more computationally expensive constraint due to its generality and the fact that it may call on external functions, it has been optimized in particular by employing function rewriting and runtime compilation.

Moreover, we implemented various output formats to avoid expensive rearrangements to different formats. By providing output formats that are close to the internal representation described in Section 3.4, potentially expensive rearrangement of the structure in which solutions are outputted by the solver is mitigated.

Finally, we introduced a parser for constraints written in string format, which has three important benefits: to apply the more efficient built-in constraints instead of generic functions where possible, to break down constraints into the smallest subsets of variables, thereby aiding the constraints solver, and to provide all of this without requiring users to write their constraints in a complex format that requires understanding how the solvers work.

To address the latter benefit first, most solvers and tuners require specific function calls or a form of domain-specific language when defining the constraints (as will be encountered in Section 4.1). 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 in Python-evaluable string format, which is then automatically optimized by parsing. This usage of Python-evaluable strings has various benefits, as they are both familiar to the user as Python is already the interface language and rewritable for the parser, allowing the application of built-in constraints instead of generic functions and the decomposition of constraints into subsets.

In particular, the automatic reduction of constraints into the smallest subsets can be important in scalability and efficiency. Constraints can be written as compound statements, e.g. $3 \leq X \cdot Y < 9 \leq Z$ where X, Y, and Z are tunable parameters with numerical values. This type of compound statement can be expected from users unfamiliar with the intrinsics of constraint solving. However, constraints are usually not evaluated by a CSP solver until values for all the involved parameters are at least partially resolved, resulting

[5] https://pypistats.org/packages/python-constraint

in subpar performance in the case of compound statements. By automatically breaking down the constraint into multiple constraints with fewer involved variables (e.g. for the given example $[3 \leq X \cdot Y, X \cdot Y < 9, 9 \leq Z]$, which can be written in built-in constraints as *[(MinProd(3), [x, y]), (MaxProd(9-1), [x, y]), (MinProd(9), [z])]*), a value for either X, Y, or Z can be enough to discard configurations not meeting the constraint early on, and there is a greater chance that built-in constraints can be applied.

3.4 Search space Representation

With the optimized resolution of constraints to efficiently construct search spaces implemented, a search space should be used by search strategies and initial sampling methods to obtain information on the available configurations and bounds of the space. This is implemented in the *SearchSpace* class available in Kernel Tuner, which provides various views and mappings on the configurations in the search space and optionally keeps track of evaluated configurations. For example, the mutation step in Genetic Algorithms requires selecting only valid neighbors with 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. The *SearchSpace* class has various 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, which in contrast to the initial situation where strategies would implement these operations individually, enables reuse in the modular architecture.

4 Evaluation

In this section, the advancements presented in Section 3 are evaluated to determine their performance and scalability impact using a case study with various applications. First, we discuss the way current state-of-the-art solvers are used to compare against in Section 4.1. Following this, we evaluate the solvers on a collection of synthetically generated search spaces to assess scalability differences between solvers under various search space characteristics in Section 4.2. Finally, we evaluate the solvers on a variety of real-world tests to indicate actual performance in Section 4.3.

The evaluations in this work are performed on the sixth generation DAS VU-cluster [3] using an NVIDIA A4000 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. While none of the tested solvers use the GPU, we used this GPU node to obtain an environment as similar as possible to real-world GPU auto-tuning. For all tests performed, the results of each solver were validated against a brute-forced solution of the search space.

4.1 Comparison against state-of-the-art

To provide additional reference on the performance in this evaluation, the results are compared to the state-of-the-art in auto-tuning search space construction: Auto-Tuning Framework (ATF) [40], which specifically focuses on large optimization spaces with interdependent parameters. ATF has two independent implementations, in C++ and Python, which are both compared against. 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.

For the majority of the discussed solvers, the notation of tunable parameters and constraints is largely separated (e.g. a user first defines the tunable parameters and values, then defines the constraints to apply). However, both implementations of ATF have a notation that combines the definition of tunable parameters, values, and constraints into one statement. As a result of this, constraints can only reference tunable parameters that have been previously defined. 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. These parsers take the aforementioned 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 using GCC 9.4.0 and includes the optimization commands recommended by the ATF documentation. The exact implementation of this evaluation is available online.

In addition, we compare against PySMT at version 0.9.6 using the Microsoft Z3 solver to evaluate differences in scalability for solvers without support for resolving all solutions, as described in Section 3.2. The Z3 theorem prover is developed by Microsoft for software verification and analysis [12], and is the winner of the 3rd Annual Satisfiability Modulo Theories Competition (SMT-COMP) [5]. Similarly to ATF, we have written a parser to use PySMT-specific operations where applicable.

4.2 Synthetic Tests

The following synthetic tests provide a better understanding of how search space characteristics influence the construction time. We have generated a set of search spaces with a varying number of dimensions (between 2 and 5), target Cartesian sizes (with $\{1 \times 10^4, 2 \times 10^4, 5 \times 10^4, 1 \times 10^5, 2 \times 10^5, 5 \times 10^5, 1 \times 10^6\}$), 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, the goal of these in total 112 synthetic search spaces is to gain insight into which of these factors has the greatest effect on performance, and which solution 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. 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. To prevent an unfair advantage to solvers optimizing for a limited number of dominant dimensions, this number of values per dimension v is kept approximately uniform. 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. $5.8 \longrightarrow 5$, $5.2 \longrightarrow 6$) to be closer to the desired Cartesian size. A list of

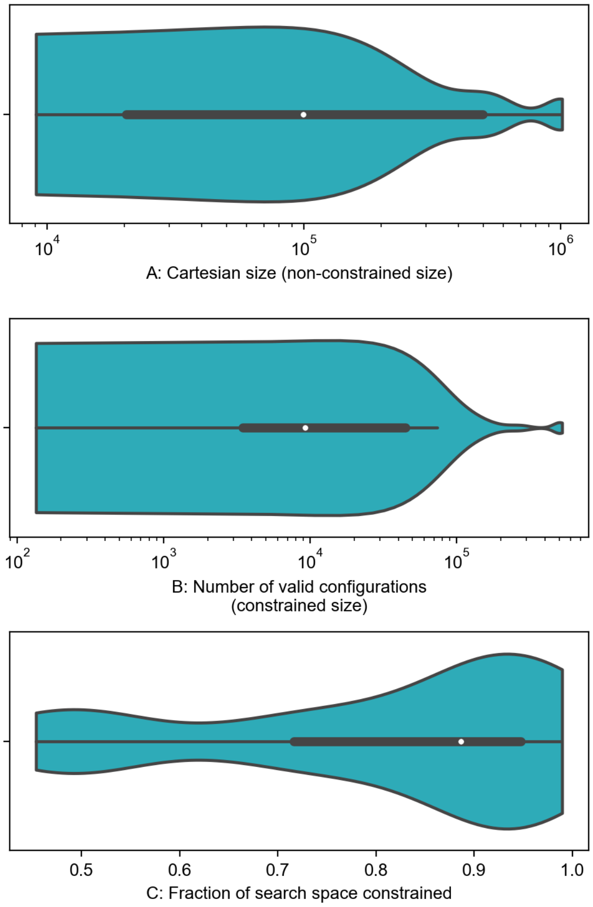

Figure 2: Characteristics of the 112 synthetic search spaces

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 dispersion of the resulting 112 search spaces over three characteristics: the Cartesian size, the number of valid configurations, and the fraction of the search space constrained (the number of valid configurations relative to the total Cartesian size). As the application of the constraints does not affect the already uniformly distributed number of parameters, this characteristic is not included in the figure. Figure 2A shows the actual Cartesian size, revealed to be in line with the set of target values used. Figure 2B depicts the number of valid configurations, which follows a distribution similar to the Cartesian size of Figure 2A, yet covering a wider range on the lower end, also demonstrating the general effect of constraints on the difference between the Cartesian size and actual number of configurations. 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. This indicates that the resulting search spaces provide a wide range the variations in sparsity.

The performance on the 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 3F barplot.

It is noteworthy how in Figure 3A there appears to be an approximately linear correlation between performance and the number of configurations, which is not as evident in the other characteristics in Figure 3B, Figure 3D, and Figure 3E. Furthermore, Figure 3A illustrates the relationship between the number of valid configurations (constrained size) and the time required to construct the search space. The *optimized* solver consistently achieves the lowest execution times, with performance improving by several orders of magnitude compared to the *brute force*, *ATF*, and *pyATF* solvers. The *pyATF* solver exhibits the highest execution times, particularly for larger search spaces.

Figure 3B examines the effect of the Cartesian size on execution time, where a general trend of increasing computation time with larger Cartesian sizes can be seen. Interestingly, the *ATF* and *pyATF* solvers in some cases do not conform to this trend, creating a large variance in their performances.

The reason for this appears to be found in Figure 3D, which shows the relationship between the fraction of the sparsity of a search space and execution time. Both *ATF* and *pyATF* appear to be severely optimized for very sparse search spaces, in contrast to the other solvers.

Figure 3C presents the execution time distributions of the solvers as a continuous probability density curve using a kernel density estimate (KDE). Particularly noteworthy is the bimodality demonstrated by *ATF* and *pyATF*, which is likely caused by the sensitivity to search space sparsity.

Observing Figure 3E, it appears that there is no strong correlation between solver performance and the number of tunable parameters.

Figure 3F summarizes the overall performance of each solver in a bar chart. The optimized method achieves a 96x speedup over the brute-force method (4.75 seconds versus 455.3 seconds). In contrast, pyATF takes considerably longer than the brute force method on these search spaces.

To demonstrate the scalability of using a traditional solver that requires adding the previous solution as a constraint and iterating over the solutions in this way, as described in Section 3.2, Figure 4 compares PySMT using the Z3 solver to the other implemented solutions. To make executing this experiment feasible, we reduced the size of the generated synthetic search spaces by one order of magnitude in this instance. As seen in Figure 4, PySMT performs poorly relative to both brute force and the optimized method. As expected, this difference increases as the size of the search space increases, demonstrating the infeasibility of this approach when many valid configurations are present. Even with 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 takes about as long to solve the largest search spaces as PySMT with the Z3 solver takes to solve the smallest search spaces. PySMT with the Z3 solver will not be included in the remainder of the evaluation as it is infeasible to evaluate the large search spaces.

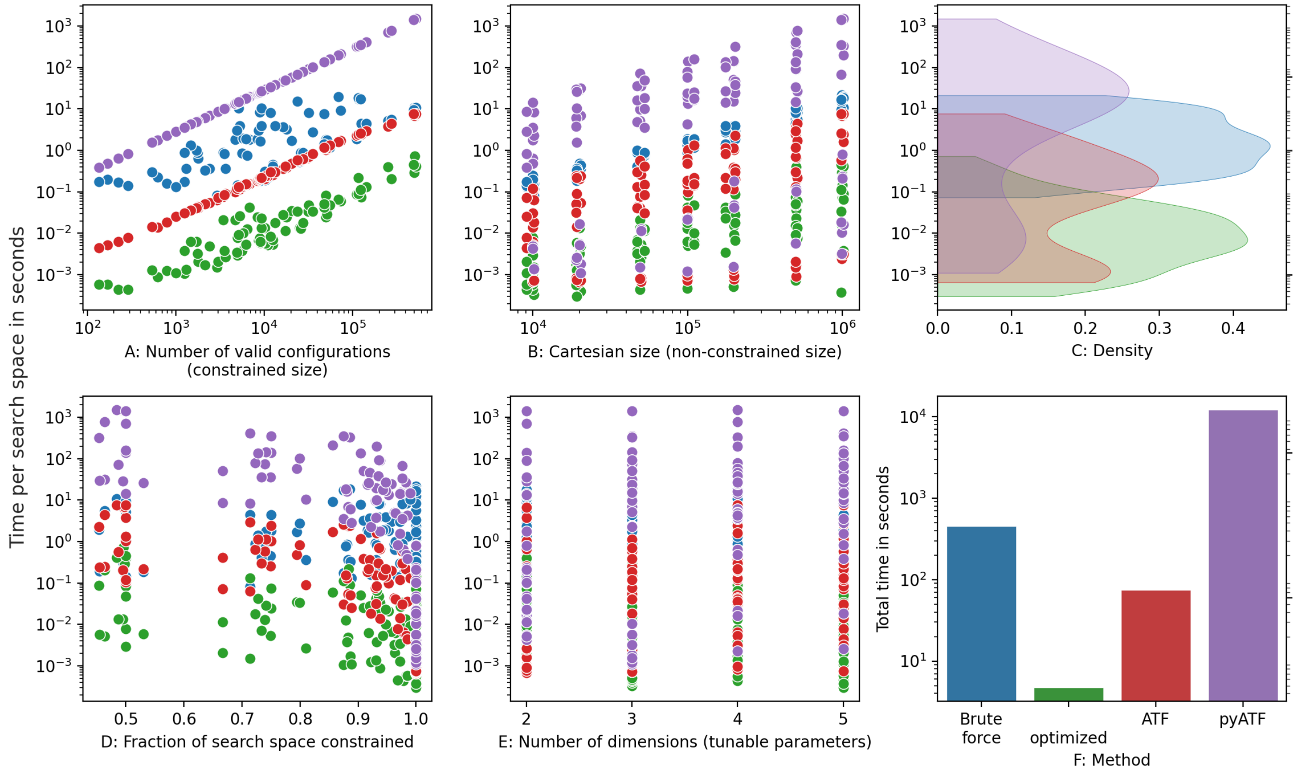

Figure 3: Search space construction performance on synthetic tests. Lower times are better. Colors correspond to Figure 3F barplot methods.

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
Dedispersion	22272	11130	8	3	2	1 - 29	49.973
ExpDist	9732096	294000	10	4	2	1 - 11	3.021
Hotspot	22200000	349853	11	5	3.8	1 - 37	1.576
GEMM	663552	116928	17	8	3.25	1 - 4	17.622
Microhh	1166400	138600	13	8	2.375	1 - 10	11.883
ATF PRL 2x2	36864	1200	20	14	2.429	1 - 3	3.255
ATF PRL 4x4	9437184	10800	20	14	2.429	1 - 4	0.114
ATF PRL 8x8	2415919104	48720	20	14	2.429	1 - 8	0.002
Mean	*307322534*	*121403*	*14.875*	*8.75*	*2.589*	*1 - 13.25*	*10.93*

Table 1: Overview of the basic characteristics of the real-world search spaces and the mean values for each of the columns.

4.3 Real-world Tests

For the real-world tests, we have used the three largest search spaces in the Benchmark suite for Auto-Tuners (BAT) [25]. These are *Dedispersion*, *Hotspot*, and *ExpDist*. In addition, we use the relatively large search spaces of the commonly used General Matrix Multiplication kernel *(GEMM)* [35] and *MicroHH* computational fluid dynamics kernel [49]. To provide a fair comparison to ATF, the Probabilistic Record Linkage (*PRL*) kernel used by the ATF paper [42] is used as well, resulting in another three search spaces for a total of eight real-world search spaces. The characteristics of the real-world search spaces are displayed in Table 1. Descriptions of each of these kernels and their search spaces are given in Sections 4.3.1 to 4.3.6, before the results are discussed in Section 4.3.7.

4.3.1 Dedispersion. The Dedispersion kernel presented in [25] 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. In this context, the signal at the highest frequency f_h arrives at time t_x, whereas lower frequencies emitted simultaneously arrive at $t_x + k$, where k represents the delay in seconds and is calculated using the following equation:

$$k \approx 4150 \times DM \times \left(\frac{1}{f_i^2} - \frac{1}{f_h^2} \right) \tag{1}$$

The kernel takes as input time-domain samples across multiple frequency channels and produces dedispersed samples for a range

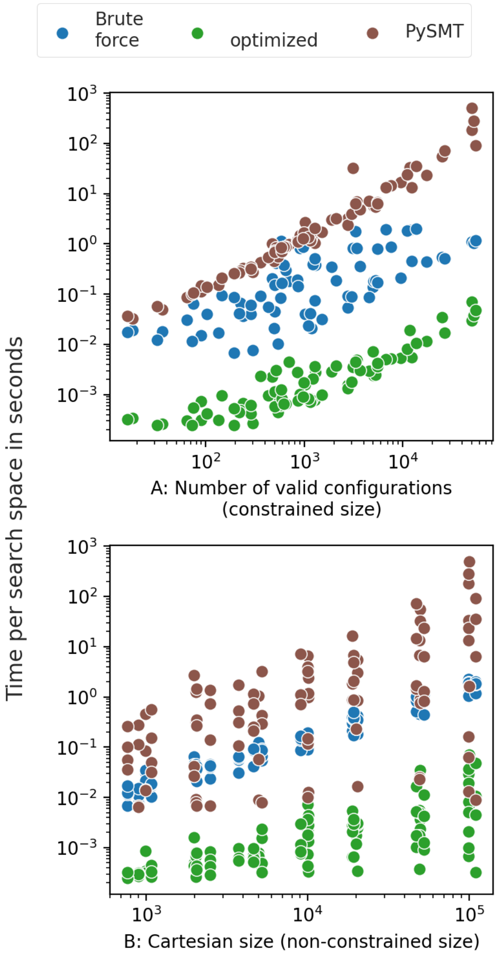

Figure 4: Search space construction performance of PySMT on synthetic tests (reduced search spaces size).

of DM values. During the iteration over frequency channels, threads process multiple time samples and dispersion measures in parallel. Comparing the Dedispersion search space to the other real-world search spaces tested on in Table 1, 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.

4.3.2 *ExpDist.* The ExpDist kernel described in [25] is utilized in a localization microscopy application that performs template-free particle fusion by integrating multiple observations into a single super-resolution reconstruction [23]. During the registration process, the ExpDist kernel is repeatedly invoked to evaluate the alignment of two particles. The distance between particles t and m, given a registration M, is calculated as follows:

$$D = \sum_{i=1}^{K_t} \sum_{j=1}^{K_m} \exp\left(-\frac{\|\vec{x}_{t,i} - M(\vec{x}_{m,j})\|^2}{2\sigma^2}\right) \tag{2}$$

The kernel operates directly on the individual localizations ($\vec{x_t}$ and $\vec{x_m}$) in each particle rather than pixelated images, accounting for uncertainties in the localizations (σ). 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 1.

4.3.3 *Hotspot.* The Hotspot kernel in [25] is part of a thermal simulation application used for estimating 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 entire 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.

4.3.4 *GEMM.* Generalized dense matrix-matrix multiplication ~~(GEMM)~~ is a fundamental operation in the BLAS linear algebra library and is widely used across various application domains. GEMM is known for its high performance on GPU hardware and has frequently served as a benchmark in studies of GPU code optimization [27, 36, 43]. In this evaluation, we utilize the GEMM kernel from CLBlast [35], 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 were set to 2048 × 2048, resulting in a relatively dense search space.

4.3.5 *MicroHH.* The computational fluid dynamics kernel of [49] 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[6] of [21]. Looking at Table 1, 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.

4.3.6 *ATF PRL.* The Probabilistic Record Linkage (PRL) kernel used in [42] is a parallelized implementation of an algorithm that is commonly used in data mining, intending to identify data records referring to the same real-world entity. In this kernel, the input sizes determine the size of the search space. Given that the brute force resolution of this search space with input sizes 8x8 took ~27 hours, it was not feasible to brute force beyond this 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,

[6] https://github.com/stijnh/microhh/blob/991c2d2407042edc0e3301d23137f85d0a291c98/kernel_tuner/helpers.py#L116-L144

and 8x8. It is notable that while their Cartesian sizes are the largest of the real-world test set, the search spaces are very sparse.

4.3.7 Results. Figure 5 presents the search space construction performance across the eight real-world benchmarks for five different constraint solver methods: *Brute force*, *original*, *optimized*, *ATF*, and *pyATF*. To determine the impact of the optimizations, the *original* method denotes the use of *python-constraint* before the optimizations described in Section 3.3, whereas the *optimized* method includes the optimizations of Section 3.3 as before in Section 4.2.

Figure 5A and Figure 5B illustrate the relationship between search space size and solver performance. In general, larger constrained search spaces (A) and Cartesian sizes (B) result in increased search times, particularly for brute-force methods. The optimized solver consistently achieves the lowest execution times across all problem sizes, demonstrating its efficiency. ATF and pyATF show a similar trend but with higher execution times compared to the optimized solver, particularly for larger spaces. The original solver exhibits significantly higher execution times than the optimized version but remains lower than brute-force 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* method is one order of magnitude faster than the *brute force* methods, both methods have very similar distributions. A clear trend emerges from this plot, where the optimized solver has the best average performance and the least variability.

In Figure 5D, the relation between how constrained a search space is and solver performance is displayed. In contrast to the synthetic tests in Section 4.2, the correlation between solver performance and the sparsity of the search space is not as clear. Nevertheless, it can be seen that the *ATF* performance is again influenced by the sparsity, as for fraction > 0.9 *ATF* performance is better than the *original* solver, as opposed to for fraction ≤ 0.9, where at fraction $\simeq 0.5$ *original* even outperforms *ATF*.

Similar to what was observed in Section 4.2, 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. 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 detailed in Table 1, 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, as expected the most computationally expensive, taking almost a full day to resolve all search spaces. The original solver, though faster than brute force, remains significantly less efficient than the optimized solver. The optimized solver considerably outperforms the others, while ATF and pyATF provide intermediate performance levels. These performance differences are even more pronounced than in the synthetic tests: the 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.

Overall, it is noteworthy that our *optimized* method consistently outperforms any alternative on all of the search spaces. These findings emphasize the advantages of the optimized solver in efficiently handling large and complex search spaces.

5 Conclusions

This paper introduces a novel approach to constructing auto-tuning search spaces for GPU kernels using an optimized Constraint Satisfaction Problem (CSP) solver, addressing the challenges posed by the complexity of auto-tuning and large search spaces. Our contributions, implemented in the open-source Kernel Tuner and python-constraint packages, substantially outperform state-of-the-art methods in terms of search space construction speed, enabling the exploration of previously unattainable problem scales in auto-tuning and related domains.

Through rigorous evaluation, we demonstrated that our optimized CSP-based approach reduces construction time by orders of magnitude, even for search spaces with billions of possible combinations. On average over the evaluated search spaces, our optimized method is three orders of magnitude faster than brute-forcing, two orders of magnitude faster than the unoptimized CSP solver, and one order of magnitude faster than the state-of-the-art in search space construction. This optimized method of search space construction reduces the search space construction time of large search spaces to such a degree that it is no longer a substantial factor in the tuning process overhead, confirming the efficiency of this new approach to search space construction. This breakthrough allows researchers and developers to more effectively harness the performance potential of modern GPUs while reducing the computational overhead associated with auto-tuning.

Kernel Tuner can be installed from the Python Package Index with `pip install kernel-tuner`, and python-constraint can be installed with `pip install python-constraint2`. Both Kernel Tuner and python-constraint are open-source software, and we welcome contributions. For more information, please visit the Kernel Tuner[7] and python-constraint[8] repositories.

Acknowledgment: The CORTEX project has received funding from the Dutch Research Council (NWO) in the framework of the NWA-ORC Call (file number NWA.1160.18.316).

References

[1] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. 2014. OpenTuner: An extensible framework for program autotuning. *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)* (2014), 303–315. https://doi.org/10.1145/2628071.2628092

[2] Amir H. Ashouri, William Killian, John Cavazos, Gianluca Palermo, and Cristina Silvano. 2019. A Survey on Compiler Autotuning using Machine Learning. *Comput. Surveys* 51, 5 (Sept. 2019), 1–42. https://doi.org/10.1145/3197978

[3] 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* 49, 05 (May 2016), 54–63. https://doi.org/10.1109/MC.2016.127 Place: Los Alamitos, CA, USA Publisher: IEEE Computer Society.

[4] Prasanna Balaprakash, Jack Dongarra, Todd Gamblin, Mary Hall, Jeffrey K. Hollingsworth, Boyana Norris, and Richard Vuduc. 2018. Autotuning in High-Performance Computing Applications. *Proc. IEEE* 106, 11 (Nov. 2018), 2068–2083. https://doi.org/10.1109/JPROC.2018.2841200

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

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

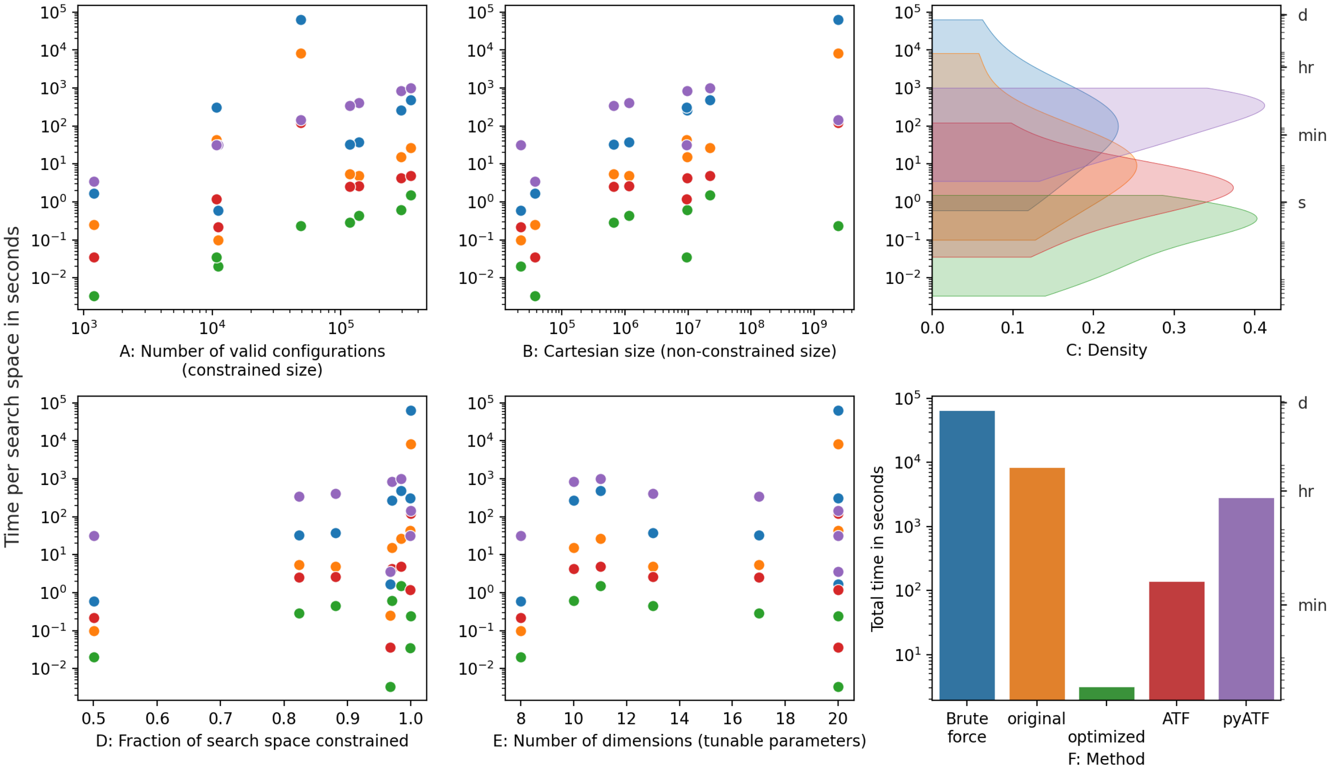

Figure 5: Search space construction performance on real-world tests. Lower times are better. Colors correspond to Figure 5F barplot methods.

[5] Clark Barrett, Morgan Deters, Albert Oliveras, and Aaron Stump. 2008. Design and results of the 3rd annual satisfiability modulo theories competition (SMT-COMP 2007). *International Journal on Artificial Intelligence Tools* 17, 04 (2008), 569–606. Publisher: World Scientific.

[6] Clark Barrett, Roberto Sebastiani, Sanjit Seshia, and Cesare Tinelli. 2008. Satisfiability Modulo Theories. Frontiers in artificial intelligence and applications, vol. 185, ch. 26. In *Handbook of Satisfiability*. Frontiers in Artificial Intelligence and Applications, Vol. 185. IOS Press, 825–885.

[7] Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn, and Kurt Smith. 2011. Cython: The Best of Both Worlds. *Computing in Science & Engineering* 13, 2 (March 2011), 31–39. https://doi.org/10.1109/MCSE.2010.118 Conference Name: Computing in Science & Engineering.

[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.* IOS Press, NLD.

[9] Nikolaj Bjørner, Leonardo de Moura, Lev Nachmanson, and Christoph M. Wintersteiger. 2019. Programming Z3. In *Engineering Trustworthy Software Systems: 4th International School, SETSS 2018, Chongqing, China, April 7–12, 2018, Tutorial Lectures*, Jonathan P. Bowen, Zhiming Liu, and Zili Zhang (Eds.). Springer International Publishing, Cham, 148–201. https://doi.org/10.1007/978-3-030-17601-3_4

[10] Sally C. Brailsford, Chris N. Potts, and Barbara M. Smith. 1999. Constraint satisfaction problems: Algorithms and applications. *European Journal of Operational Research* 119, 3 (Dec. 1999), 557–581. https://doi.org/10.1016/S0377-2217(98)00364-6

[11] TT Dao and J Lee. 2017. An auto-tuner for OpenCL work-group size on GPUs. *IEEE Transactions on Parallel and Distributed Systems* 29, 2 (2017), 283 – 296. https://ieeexplore.ieee.org/abstract/document/8048544/ Publisher: IEEE.

[12] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on tools and algorithms for the construction and analysis of systems*. Springer, 337–340.

[13] Pablo de Oliveira Castro, Eric Petit, Jean Christophe Beyler, and William Jalby. 2012. ASK: Adaptive sampling kit for performance characterization. In *Euro-par 2012 parallel processing*, Christos Kaklamanis, Theodore Papatheodorou, and Paul G. Spirakis (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 89–101.

[14] T. L. Falch and A. C. Elster. 2015. Machine learning based auto-tuning for enhanced OpenCL performance portability. In *2015 IEEE international parallel and distributed processing symposium workshop*. IEEE, Hyderabad, India, 1231–1240. https://doi.org/10.1109/IPDPSW.2015.85

[15] Jiří Filipovič, Jana Hozzová, Amin Nezarat, Jaroslav Ol'ha, and Filip Petrovič. 2022. Using hardware performance counters to speed up autotuning convergence on GPUs. *J. Parallel and Distrib. Comput.* 160 (Feb. 2022), 16–35. https://doi.org/10.1016/j.jpdc.2021.10.003

[16] Jiři Filipovič, Filip Petrovič, and Siegfried Benkner. 2017. Autotuning of OpenCL kernels with global optimizations. In *Proceedings of the 1st workshop on AutotuniNg and adaptivity AppRoaches for energy efficient HPC systems (Andare '17)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3152821.3152877 Number of pages: 6 Place: Portland, OR, USA tex.articleno: 2.

[17] Matteo Frigo and Steven G Johnson. 1998. FFTW: An adaptive software architecture for the FFT. In *Acoustics, speech and signal processing, 1998. Proceedings of the 1998 IEEE international conference on*, Vol. 3. IEEE, 1381–1384.

[18] Tias Guns. 2019. Increasing modeling language convenience with a universal n-dimensional array, CPpy as python-embedded example. In *Proceedings of the 18th workshop on Constraint Modelling and Reformulation at CP (Modref 2019)*, Vol. 19. Modref 2019. https://github.com/CPMpy/cpmpy

[19] A. Hartono, B. Norris, and P. Sadayappan. 2009. Annotation-based empirical performance tuning using Orio. In *2009 IEEE international symposium on parallel distributed processing*. IEEE, Rome, Italy„ 1–11. https://doi.org/10.1109/IPDPS.2009.5161004

[20] Stijn Heldens, Pieter Hijma, Ben Van Werkhoven, Jason Maassen, Adam S. Z. Belloum, and Rob V. Van Nieuwpoort. 2021. The Landscape of Exascale Research: A Data-Driven Literature Analysis. *Comput. Surveys* 53, 2 (March 2021), 1–43. https://doi.org/10.1145/3372390

[21] Stijn Heldens and Ben van Werkhoven. 2023. Kernel Launcher: C++ Library for Optimal-Performance Portable CUDA Applications. https://doi.org/10.48550/arXiv.2303.12374 arXiv:2303.12374 [cs].

[22] Erik Orm Hellsten, Artur Souza, Johannes Lenfers, Rubens Lacouture, Olivia Hsu, Adel Ejjeh, Fredrik Kjolstad, Michel Steuwer, Kunle Olukotun, and Luigi Nardi.

2024. BaCO: a fast and portable bayesian compiler optimization framework. In *Proceedings of the 28th ACM international conference on architectural support for programming languages and operating systems, volume 4 (Asplos '23)*. Association for Computing Machinery, New York, NY, USA, 19–42. https://doi.org/10.1145/3623278.3624770 Number of pages: 24 Place: Vancouver, BC, Canada.

[23] Hamidreza Heydarian, Florian Schueder, Maximilian T Strauss, Ben van Werkhoven, Mohamadreza Fazel, Keith A Lidke, Ralf Jungmann, Sjoerd Stallinga, and Bernd Rieger. 2018. Template-free 2D particle fusion in localization microscopy. *Nature methods* 15, 10 (2018), 781–784. Publisher: Nature Publishing Group.

[24] Pieter Hijma, Stijn Heldens, Alessio Sclocco, Ben Van Werkhoven, and Henri E. Bal. 2023. Optimization Techniques for GPU Programming. *Comput. Surveys* 55, 11 (Nov. 2023), 1–81. https://doi.org/10.1145/3570638

[25] Jacob O. Tørring, Ben van Werkhoven, Filip Petrovič, Floris-Jan Willemsen, Jiří Filipovič, and Anne C. Elster. 2023. Towards a Benchmarking Suite for Kerneltuners (accepted at iWAPT 2023). https://www.overleaf.com/project/638e0716ca3dc21d79f564ba

[26] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *Nature* 521, 7553 (May 2015), 436–444. https://doi.org/10.1038/nature14539

[27] Yinan Li, Jack Dongarra, and Stanimire Tomov. 2009. A note on auto-tuning GEMM for GPUs. In *Computational science–ICCS 2009: 9th international conference baton rouge, la, USA, may 25-27, 2009 proceedings, part I 9*. Springer, 884–892.

[28] Yang Liu, Wissam M. Sid-Lakhdar, Osni Marques, Xinran Zhu, Chang Meng, James W. Demmel, and Xiaoye S. Li. 2021. GPTune: Multitask Learning for Autotuning Exascale Applications. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '21)*. Association for Computing Machinery, New York, NY, USA, 234–246. https://doi.org/10.1145/3437801.3441621 event-place: Virtual Event, Republic of Korea.

[29] Google LLC. 2015. ortools: Google OR-Tools python libraries and modules. https://developers.google.com/optimization/

[30] Milo Lurati, Stijn Heldens, Alessio Sclocco, and Ben Van Werkhoven. 2024. Bringing Auto-Tuning to HIP: Analysis of Tuning Impact and Difficulty on AMD and Nvidia GPUs. In *Euro-Par 2024: Parallel Processing*, Jesus Carretero, Sameer Shende, Javier Garcia-Blas, Ivona Brandic, Katzalin Olcoz, and Martin Schreiber (Eds.). Vol. 14801. Springer Nature Switzerland, Cham, 91–106. https://doi.org/10.1007/978-3-031-69577-3_7 Series Title: Lecture Notes in Computer Science.

[31] Fábián Tamás László. 2013. satispy: An interface to SAT solver tools (like minisat). https://github.com/netom/satispy/

[32] Sanskar Mani. 2020. CSP-Solver: Library to solve Constraint satisfation problems. https://github.com/LezendarySandwich/Generic-CSP-Solver

[33] Luigi Nardi, Artur Souza, David Koeplinger, and Kunle Olukotun. 2019. HyperMapper: a Practical Design Space Exploration Framework. In *2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 425–426. https://doi.org/10.1109/MASCOTS.2019.00053 ISSN: 2375-0227.

[34] Gustavo Niemeyer. 2005. python-constraint: python-constraint is a module implementing support for handling CSPs (Constraint Solving Problems) over finite domain. https://github.com/python-constraint/python-constraint

[35] Cedric Nugteren. 2018. CLBlast: A tuned OpenCL BLAS library. In *Proceedings of the international workshop on OpenCL (IWOCL '18)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3204919.3204924 Number of pages: 10 Place: Oxford, United Kingdom tex.articleno: 5.

[36] C Nugteren and V Codreanu. 2015. CLTune: A generic auto-tuner for OpenCL kernels. *2015 IEEE 9th International …* (2015). https://ieeexplore.ieee.org/abstract/document/7328205/ Publisher: IEEE.

[37] Eric Papenhausen and Klaus Mueller. 2018. Coding Ants: Optimization of GPU code using ant colony optimization. *Computer Languages, Systems & Structures* 54 (2018), 119 – 138. https://doi.org/10.1016/j.cl.2018.05.003

[38] Dimitri Justeau-Allaire Prud'homme, Charles. 2022. pychoco: Python bindings to the Choco Constraint Programming solver.

[39] A Rasch and S Gorlatch. 2018. ATF: A generic directive-based auto-tuning framework. *Concurrency and Computation: Practice and Experience* (2018). https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4423 Publisher: Wiley Online Library.

[40] Ari Rasch, Michael Haidl, and Sergei Gorlatch. 2017. ATF: A Generic Auto-Tuning Framework. In *2017 IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. 64–71. https://doi.org/10.1109/HPCC-SmartCity-DSS.2017.9

[41] Ari Rasch, Richard Schulze, Michel Steuwer, and Sergei Gorlatch. 2021. Efficient auto-tuning of parallel programs with interdependent tuning parameters via auto-tuning framework (ATF). *ACM Trans. Archit. Code Optim.* 18, 1 (Jan. 2021). https://doi.org/10.1145/3427093 Number of pages: 26 Place: New York, NY, USA Publisher: Association for Computing Machinery tex.articleno: 1 tex.issue_date: January 2021.

[42] Ari Rasch, Richard Schulze, Michel Steuwer, and Sergei Gorlatch. 2021. Efficient auto-tuning of parallel programs with interdependent tuning parameters via auto-tuning framework (ATF). *ACM Trans. Archit. Code Optim.* 18, 1 (Jan. 2021). https://doi.org/10.1145/3427093 Number of pages: 26 Place: New York, NY, USA Publisher: Association for Computing Machinery tex.articleno: 1 tex.issue_date: March 2021.

[43] Shane Ryoo, Christopher I. Rodrigues, Sam S. Stone, Sara S. Baghsorkhi, Sain-Zee Ueng, John A. Stratton, and Wen-mei W. Hwu. 2008. Program optimization space pruning for a multithreaded gpu. In *Proceedings of the 6th annual IEEE/ACM international symposium on code generation and optimization (Cgo '08)*. Association for Computing Machinery, New York, NY, USA, 195–204. https://doi.org/10.1145/1356058.1356084 Number of pages: 10 Place: Boston, MA, USA.

[44] Ilan Schnell. 2013. pycosat: bindings to picosat (a SAT solver). https://github.com/ContinuumIO/pycosat

[45] Richard Schoonhoven, Ben van Werkhoven, and Kees Joost Batenburg. 2022. Benchmarking optimization algorithms for auto-tuning GPU kernels. *IEEE Transactions on Evolutionary Computation* (2022), 1–1. https://doi.org/10.1109/TEVC.2022.3210654 arXiv:2210.01465 [cs].

[46] Richard Schoonhoven, Bram Veenboer, Ben Van Werkhoven, and K. Joost Batenburg. 2022. Going green: optimizing GPUs for energy efficiency through model-steered auto-tuning. *2022 IEEE/ACM International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)* (Nov. 2022), 48–59. https://doi.org/10.1109/PMBS56514.2022.00010 Conference Name: 2022 IEEE/ACM International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS) ISBN: 9781665451857 Place: Dallas, TX, USA Publisher: IEEE.

[47] Mirko Stojadinović and Filip Marić. 2014. meSAT: multiple encodings of CSP to SAT. *Constraints* 19, 4 (Oct. 2014), 380–403. https://doi.org/10.1007/s10601-014-9165-7

[48] PySMT Team. 2022. PySMT: A solver-agnostic library for SMT Formulae manipulation and solving. http://www.pysmt.org

[49] Chiel C Van Heerwaarden, Bart JH Van Stratum, Thijs Heus, Jeremy A Gibbs, Evgeni Fedorovich, and Juan Pedro Mellado. 2017. MicroHH 1.0: A computational fluid dynamics code for direct numerical simulation and large-eddy simulation of atmospheric boundary layer flows. *Geoscientific Model Development* 10, 8 (2017), 3145–3165. Publisher: Copernicus GmbH.

[50] Ben van Werkhoven. 2019. Kernel Tuner: A search-optimizing GPU code auto-tuner. *Future Generation Computer Systems* 90 (Jan. 2019), 347–358. https://doi.org/10.1016/j.future.2018.08.004

[51] Ben van Werkhoven, Willem Jan Palenstijn, and Alessio Sclocco. 2020. Lessons learned in a decade of research software engineering gpu applications. In *International conference on computational science*. Springer, 399–412.

[52] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral parallel code generation for CUDA. *ACM Trans. Archit. Code Optim.* 9, 4 (Jan. 2013). https://doi.org/10.1145/2400682.2400713 Number of pages: 23 Place: New York, NY, USA Publisher: Association for Computing Machinery tex.articleno: 54 tex.issue_date: January 2013.

[53] B Videau, K Pouget, L Genovese, T Deutsch, D Komatitsch, F Desprez, and JF Mehau. 2017. BOAST: A metaprogramming framework to produce portable and efficient computing kernels for HPC applications. *The International Journal of High Performance Computing Applications* (2017). https://journals.sagepub.com/doi/abs/10.1177/1094342017718068 Publisher: SAGE Publications.

[54] R Clint Whaley, Antoine Petitet, and Jack J Dongarra. 2001. Automated empirical optimizations of software and the ATLAS project. *Parallel Comput.* 27, 1-2 (2001), 3–35. Publisher: Elsevier.

[55] Floris-Jan Willemsen, Rob van Nieuwpoort, and Ben van Werkhoven. 2021. Bayesian Optimization for auto-tuning GPU kernels. In *2021 International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. IEEE, 106–117. https://doi.org/10.1109/PMBS54543.2021.00017

[56] Xingfu Wu, Prasanna Balaprakash, Michael Kruse, Jaehoon Koo, Brice Videau, Paul Hovland, Valerie Taylor, Brad Geltz, Siddhartha Jana, and Mary Hall. 2024. ytopt: Autotuning Scientific Applications for Energy Efficiency at Large Scales. *Concurrency and Computation: Practice and Experience* (Oct. 2024), e8322. https://doi.org/10.1002/cpe.8322