Efficient Auto-Tuning of Parallel Programs with Interdependent Tuning Parameters via Auto-Tuning Framework (ATF)

ARI RASCH and RICHARD SCHULZE, University of Muenster, Germany
MICHEL STEUWER, University of Edinburgh, United Kingdom
SERGEI GORLATCH, University of Muenster, Germany

Auto-tuning is a popular approach to program optimization: it automatically finds good configurations of a program's so-called tuning parameters whose values are crucial for achieving high performance for a particular parallel architecture and characteristics of input/output data. We present three new contributions of the Auto-Tuning Framework (ATF), which enable a key advantage in *general-purpose auto-tuning*: efficiently optimizing programs whose tuning parameters have *interdependencies* among them. We make the following contributions to the three main phases of general-purpose auto-tuning: (1) ATF *generates* the search space of interdependent tuning parameters with high performance by efficiently exploiting parameter constraints; (2) ATF *stores* such search spaces efficiently in memory, based on a novel chain-of-trees search space structure; (3) ATF *explores* these search spaces faster, by employing a multi-dimensional search strategy on its chain-of-trees search space representation. Our experiments demonstrate that, compared to the state-of-the-art, general-purpose auto-tuning frameworks, ATF substantially improves generating, storing, and exploring the search space of interdependent tuning parameters, thereby enabling an efficient overall auto-tuning process for important applications from popular domains, including stencil computations, linear algebra routines, quantum chemistry computations, and data mining algorithms.

CCS Concepts: • **General and reference → Performance**; • **Computer systems organization → Parallel architectures**; • **Software and its engineering → Parallel programming languages**;

Additional Key Words and Phrases: Auto-tuning, parallel programs, interdependent tuning parameters

ACM Reference format:
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, Article 1 (January 2021), 26 pages.
https://doi.org/10.1145/3427093

This is a new paper, not an extension of a conference paper.

Authors' addresses: A. Rasch, R. Schulze, and S. Gorlatch, University of Muenster, Muenster, Germany; emails: {a.rasch, r.schulze, gorlatch}@uni-muenster.de; M. Steuwer, University of Edinburgh, Edinburgh, United Kingdom; email: michel.steuwer@glasgow.ac.uk.

1544-3566/2021/01-ART1
https://doi.org/10.1145/3427093

1 INTRODUCTION

High performance for parallel programs is difficult to achieve, because program code has to be optimized for different target architectures and for changing input/output characteristics (size, dimensionality, transposition layout, etc.) [63]. Typically, there are many parameters that influence a program's performance in different ways, such that finding the optimal parameter configuration is often a hardly manageable task even for experts.

Auto-tuning is a technique for automatically finding good values of a program's performance-critical parameters (a.k.a. *tuning parameters*), e.g., the number of threads and/or sizes of tiles, for a given architecture and input/output characteristics. Usually, the auto-tuning process consists of three major phases: the *generation*, *storing*, and *exploration* of the program-specific *search space*, which consists of all possible parameter configurations.

There have been several successful *special-purpose auto-tuning* approaches, with an overview in [5]. They achieve impressive results for particular application classes on particular target architectures, by taking advantage of domain-specific knowledge to efficiently generate, store, and explore the program-specific search space. Notable examples of special-purpose auto-tuners are ATLAS [69] and PATUS [12], where auto-tuning is used for optimizing linear algebra routines on CPU architectures or for high-performance stencil computations on CPUs and GPUs, respectively.

Unfortunately, the implementation of a special-purpose auto-tuner is a cumbersome task. The developer has to manually manage the generation and storing of the parameter configurations, and tailor a search technique (like genetic algorithms or simulated annealing [70]) to the parameters' search space for its automatic exploration. Special-purpose auto-tuning becomes especially challenging when parameters have *interdependencies* among them, e.g., when the value of one parameter must be divisible by the value of another parameter. Such divisibility properties are often required for tuning parameters of recent parallel applications; e.g., in order to correctly exploit the thread and memory hierarchy of modern architectures, as we discuss later in this article. Designing a special-purpose auto-tuner for such interdependent parameters requires expert knowledge and a significant implementation effort from the auto-tuning developer. We demonstrate that generating and storing the search spaces of interdependent tuning parameters is time-consuming and memory-intensive, and that the structure of such spaces significantly impacts the exploration efficiency of state-of-the-art search techniques. The demand for higher productivity in auto-tuning has been recently identified as a major research challenge in high-performance computing [5, 6].

Our work is inspired by the alternative approach, *general-purpose auto-tuning*, with the classical approaches Orio [19] and ActiveHarmony [65], followed by the current state-of-the-art frameworks OpenTuner [2] and CLTune [41], and the most recent `libtuning` [46] and KernelTuner [66] approaches. General-purpose auto-tuning aims at simplifying the auto-tuning process: the software developer specifies a program's tuning parameters by their names and ranges of possible values, and the general-purpose framework then automatically creates the corresponding special-purpose auto-tuner that generates, stores, and explores the program-specific search space.

The existing general-purpose auto-tuning approaches are efficient for many applications on a range of architectures; however, we demonstrate in this article that they still struggle with programs whose tuning parameters have interdependencies among them: the existing approaches either keep invalid configurations within their search spaces, which often hinders search techniques from finding well-performing configurations (like Orio, OpenTuner, and `libtuning`), or the approaches have difficulties with efficiently generating, storing, and exploring the search spaces of recent parallel applications that consist of only valid configurations (like ActiveHarmony, CLTune, and KernelTuner).

We present three new contributions of our Auto-Tuning Framework (ATF) to address the discussed weaknesses in state-of-the-art general-purpose auto-tuning for programs with

interdependent tuning parameters. The ATF framework was originally introduced in previous work—as both an offline [49] and an online [50] approach[1]—where its user interface and convenient usage are presented, based on prototype mechanisms for search space generation, storing, and exploration. In contrast, we introduce in this article, new contributions of ATF to each particular phase of the auto-tuning process:

(1) ATF *generates* the search space of interdependent parameters with higher performance than the current general-purpose auto-tuners by efficiently exploiting parameter constraints;
(2) ATF *stores* the generated search space of such parameter more efficiently in memory by relying on a novel chain-of-trees search space structure for representing these spaces;
(3) ATF *explores* the generated and stored space of interdependent parameters faster by employing a multi-dimensional search strategy on its chain-of-trees search space representation.

Our experiments confirm that ATF substantially improves the state of the art in general-purpose auto-tuning (including ATF's former, prototype implementation [49, 50]), based on four popular application case studies: (1) stencil computation *Gaussian Convolution*, (2) linear algebra routine *General Matrix-Matrix Multiplication*, (3) quantum chemistry computation *Coupled Cluster*, and (4) data mining algorithm *Probabilistic Record Linkage*.

The rest of the article is structured as follows. Section 2 briefly recapitulates the state of the art in general-purpose auto-tuning. Sections 3, 4, and 5 introduce our novel mechanisms for generating, storing, and exploring the search spaces of interdependent tuning parameters, and Section 6 illustrates the user interface of the ATF, which implements our novel mechanisms. Our experimental evaluation is described in Section 7. We discuss related work in Section 8, and we conclude in Section 9.

2 STATE-OF-THE-ART GENERAL-PURPOSE AUTO-TUNING APPROACHES

We briefly recapitulate the state of the art in general-purpose auto-tuning: OpenTuner and `libtuning` in Section 2.1, and CLTune and KernelTuner in Section 2.2. Classical approaches ActiveHarmony and Orio are discussed in Section 8.

2.1 Auto-Tuners Designed Toward Independent Tuning Parameters

OpenTuner and `libtuning` are auto-tuning frameworks designed and optimized toward applications, whose tuning parameters have no interdependencies among them. This restriction enables both approaches to rely on only straightforward mechanisms for search space generation, storing, and exploration: the user specifies tuning parameters by their name and range of possible values, and, per design, each possible combination of parameters' values is considered by the auto-tuner as a valid configuration within the search space. Therefore, explicitly generating and storing the entire search space is not required in these approaches, because configurations can be generated straightforwardly, on the fly, by arbitrarily combining parameters' values, and storing these spaces requires only storing parameters' ranges that have a small memory footprint. Furthermore, search techniques can be easily used for exploring these systems' search spaces: the spaces have rectangular shapes where each dimension of the space represents the range of a particular tuning parameter; this enables straightforwardly mapping the spaces to a *coordinate space*—a collection

[1]Online auto-tuning happens at runtime so that it can be based on runtime values (e.g., the input size), while offline auto-tuning works at compile time and uses static parameters only [5].

of equally sized sequences of real numbers, for which numerical search techniques are specifically designed and optimized [70].

While auto-tuning systems for independent parameters work well for many applications, they struggle with programs whose tuning parameters have interdependencies among them. This is because arbitrarily combining the values of interdependent parameters can lead to invalid parameter configurations which cannot be distinguished in OpenTuner and `libtuning` due to their inherent design (the user has to manually set a penalty value for invalid configurations, as a workaround [45]). Consequently, the tuners' spaces contain also invalid configurations which are often inefficient: we demonstrate for important parallel applications that often $> 99.999\%$ of configurations within their search spaces are invalid due to parameters' interdependencies, which hinders state-of-the-art search techniques from finding well-performing configurations.

2.2 Auto-Tuners Designed Toward Interdependent Tuning Parameters

CLTune and KernelTuner are popular auto-tuners that take interdependencies among tuning parameters into account. For this, the frameworks generate, store, and explore on the *constrained search space* which contains valid configurations only so that their search techniques do not have to struggle with invalid configurations. For example, we show in Section 7 that using the constrained search space, it is possible to find well-performing configurations for important applications (e.g., stencil computations and linear algebra routines) in a reasonable 4h of tuning time, in which search techniques explore up to 20,000 valid configurations in the constrained space. In contrast, when relying on the *unconstrained search space* which contains also invalid configurations (as in OpenTuner and `libtuning`), it is not possible to even find a valid starting point within the space—independent of the chosen search technique—in 4h exploration time (in which, e.g., OpenTuner tested up to 190,000 configurations). This is because the unconstrained search space often contains a vast amount of invalid configurations.

CLTune and KernelTuner are efficient for many applications; however, both approaches rely on the same, straightforward processes to search space generation, storing, and exploration, which are inefficient for the large spaces of recent parallel applications: the two approaches use a naive search space representation which straightforwardly enumerates all valid configurations within a one-dimensional array; we demonstrate that such a representation causes a large memory footprint and hinders the efficiency of state-of-the-art search techniques, because exploration can be performed in only one dimension. Furthermore, the two approaches generate these spaces based on a so-called *search space constraint* which has to be checked also for all invalid configurations; this is inefficient when the number of invalid configurations is large.

3 GENERATING CONSTRAINED SEARCH SPACES

We address the first phase of auto-tuning by introducing a novel generation algorithm for constrained search spaces: we recapitulate ATF's parameter constraints in Section 3.1, and we show how we efficiently exploit ATF's constraint design for fast search space generation in Section 3.2.

3.1 Parameter Constraints

A key concept in auto-tuning is a *tuning parameter*. Typically, a tuning parameter p_i is represented by a pair containing the parameter's name and a range which specifies the parameter's possible values:

$$p_i := (\langle \texttt{name} \rangle, \langle \texttt{range} \rangle).$$

ATF extends this traditional definition of a tuning parameter by adding to it a *parameter constraint*:

$$p_i := (\langle \texttt{name} \rangle, \langle \texttt{range} \rangle, \langle \texttt{constraint} \rangle).$$

A parameter constraint may be any arbitrary, unary, Boolean C++17 function [23] that takes as input an element of its parameter's range; values for which the function returns `false` are filtered out of the range.

Parameter constraints enable expressing arbitrary interdependencies among tuning parameters and consequently to avoid invalid configurations within the search space. For this, the constraint function of a parameter p_i may use in its definition all previously defined tuning parameters p_j, $j < i$, as common C++ variables that have the same type as their corresponding range values, e.g., type `int` in the case of a tuning parameter p_j whose range consists of integers. For example, in OpenCL—the standard for uniformly programming different kinds of parallel architectures—the number of threads in a group (local size) has to divide the overall number of threads (global size), and the global size usually has to be smaller than or equal to the input size N to avoid idling threads. To express this, we use the following Boolean unnamed C++ functions as constraints for the global and local size tuning parameters, where `%` denotes the modulo operator:

```
// global size parameter constraint
[]( int global_size ){ return global_size <= N; }

// local size parameter constraint
[]( int local_size ){ return global_size % local_size == 0; }
```

The constraint function of the local size parameter uses the global size parameter `global_size` in its body, thus expressing the interdependency among these two parameters. We discuss the definition and usage of parameter constraints in ATF's user interface in more detail in Section 6.

For comparison, a search space constraint in CLTune and KernelTuner that is equivalent to the ATF's two parameter constraints above is [41, 66]

```
[]( auto c ){ return c.local_size * c.k <= N; }
```

A search space constraint has to be defined as a single function (with drawbacks discussed in the next subsection). In this example, the constraint takes as input a configuration `c` comprising tuning parameters `local_size` and `k`; the `global_size` is then computed as `local_size * k`.

Note that ATF's parameter constraints are as expressive as the traditional search space constraints: a search space constraint that is equivalent to a set of parameter constraints can always be generated by combining the parameter constraints via logical and. Moreover, [49, 50] show that ATF's user interface (based on parameter constraints) provides a better user experience as compared to the interfaces of CLTune (which relies on search space constraints) and OpenTuner (which supports no constraints at all).

In the following, we show how we exploit ATF's constraint design for fast search space generation.

3.2 Algorithm for Generating Constrained Search Spaces

We first briefly present the traditional generation algorithm for constrained search spaces as used in CLTune and KernelTuner, which is based on search space constraints. Afterwards, we introduce our novel algorithm for generating constrained search spaces, which is based on parameter constraints.

Traditional Approach. Listing 1 shows as pseudocode the original search space generation algorithm of CLTune and KernelTuner (taken from [41] and [66]), which is based on a search space constraint and the traditional definition of tuning parameters. Configurations are added to the search space (line 8 in Listing 1) if the search space constraint *sc* (line 6) is satisfied. We use the C++ syntax for range-based `for`-loops, where r_i denotes the range of the i-th tuning parameter (lines 2–4).

A major drawback of the traditional approach is that the search space constraint (line 6) has to be checked at the deepest level of the loop nest, causing high search space generation time.

```
1 // iteration over tuning parameter configurations
2 for ( v_1 : r_1 )
3   ...
4   for ( v_k : r_k )
5     // checking search space constraint
6     if ( sc(v_1,...,v_k) )
7       // adding configuration to the search space
8       add_config( v_1,...,v_k );
```

Listing 1. Traditional algorithm (pseudocode) for generating constrained search spaces [41, 66].

Novel Approach. Listing 2 shows as pseudocode our optimized algorithm for generating constrained search spaces, which relies on parameter constraints, rather than on search space constraints as in the traditional approach. Compared to Listing 1, our algorithm in Listing 2 exploits ATF's constraint design to make two major optimizations: (1) generating independently and in parallel the search space parts of groups of interdependent tuning parameters, and (2) checking constraints early in the loop nest. We discuss both optimizations in the following.

```
1  // processing groups of interdependent parameters in parallel
2  parallel for ( G : {G_1,...,G_n} )
3    // processing individual interdependent parameter groups in parallel
4    parallel for ( v_1^G : r_1^G )
5      if ( pc_1^G <> (v_1^G) )
6        ...
7        parallel for ( v_{tG}^G : r_{tG}^G )
8          if ( pc_{tG}^G < n_1^G,...,n_{tG-1}^G > (v_{tG}^G) )
9            // sequential computation of a group
10           for ( v_{tG+1}^G : r_{tG+1}^G )
11             if ( pc_{tG+1}^G < n_1^G,...,n_{tG}^G > (v_{tG+1}^G) )
12               ...
13                 for ( v_{kG}^G : r_{kG}^G )
14                   if ( pc_{kG}^G < n_1^G,...,n_{kG-1}^G > (v_{kG}^G) )
15                     // adding configuration to the search space
16                     search_space.group(G).add_par( v_1^G,...,v_{tG}^G ).add_seq( v_{tG+1}^G,...,v_{kG}^G );
```

Listing 2. Novel algorithm (pseudocode) for generating constrained search spaces.

Optimization 1: In general, not all tuning parameters depend on each other. For example, in recent CPU and GPU implementations [51], e.g., for stencil computations and linear algebra routines, up to 39 parameters are used (as we discuss in Section 7); these parameters can be partitioned into differently sized groups of interdependent tuning parameters; e.g., six parameter groups in the case of stencil commutations, which comprise up to four parameters in a group. We exploit ATF's constraint design to automatically identify interdependent parameter groups: two parameters are interdependent and thus in the same group, iff one of them occurs in the syntax

tree of the other parameter's constraint function. In the following, let $G_1, \ldots, G_n$ be the disjoint grouping of interdependent parameters, where each group G comprises k^G tuning parameters: $G = \{p_1^G, \ldots, p_{k^G}^G\}$, $p_i^G = (n_i^G, r_i^G, pc_i^G < n_1^G, \ldots, n_{i-1}^G >)$, $1 \leq i \leq k^G$; here, $< n_1^G, \ldots, n_{i-1}^G >$ denotes that constraint pc_i^G may use all previously defined tuning parameters (as discussed above). In Listing 2, for each group G, we generate its corresponding part of the search space independently of the other group's parts (line 2). This breaks the deep loop nests in Listing 1, leading to significantly faster space generation, as we confirm later in our experiments. Moreover, as the groups' search space parts can be generated independently of each other, we can generate them in parallel (indicated by keyword `parallel` in line 2).

We use a further potential of parallelism in Listing 2 by generating a group G's first t^G `for`-loops also in parallel (lines 4–6 in Listing 2); here, t^G denotes an arbitrary, user-defined constant between 1 and k^G. We set t^G to a default value of $t^G = 1$, i.e., we parallelize only the first loop of the nest (line 4) because, in most cases, this is sufficient for high parallelization: the first parameter's range usually comprises more values than cores available in the target system's CPU. However, in special cases, if the first tuning parameter has a small range (e.g., a Boolean parameter), we set t^G to a higher value, and consequently parallelize more loops in the nest, in order to fully utilize the available hardware. To avoid a parallelization overhead which might be high even for $t^G = 1$ if the first parameter's range is large, our parallel implementation uses a thread pool comprising as many threads as cores are available in the target CPU.

Synchronization is not required in our parallel algorithm, because the subspaces of different groups (accessed via function `group` in line 14 of Listing 2) and the subspaces of a group G's first t^G parameters (added via `add_par`) are disjoint.

Note that in general, groups of interdependent parameters cannot be identified automatically when using a traditional search space constraint as in CLTune and KernelTuner: by design, parameters' interdependencies are defined in a single search space constraint only, thus requiring a complicated semantic analysis of the constraint.

Optimization 2: In the traditional algorithm, constraints must be checked at the deepest level of the loop nest, because the algorithm relies on a search space constraint which checks full configurations (line 6 in Listing 1). In contrast, ATF's constraint design enables checking constraints early in the loop nest, thereby avoiding iterations over entire subspaces (Listing 2, lines 5–12), which substantially accelerates search space generation.

4 STORING CONSTRAINED SEARCH SPACES

In this section, we address the second phase of auto-tuning: storing the space of configurations, which is generated according to the first phase described in Section 3. If tuning parameters have no interdependencies, as assumed by OpenTuner and `libtuning`, then their search space can be represented straightforwardly using only the ranges of the tuning parameters (as discussed in Section 2.1), because each combination of values in parameters' ranges represents a valid configuration. In contrast, representing a constrained search space in the case of interdependencies among the parameters is significantly more complex, because not all configurations of parameters are valid.

CLTune and KernelTuner address this problem of interdependent parameters by generating and storing in memory *a priori* the entire constrained search space. This allows search techniques to freely navigate over the space, as required by the techniques for high search efficiency [70]. However, both approaches store the space in a plain array of configurations, which wastes a significant amount of memory space, because many configurations share the same parameter values.

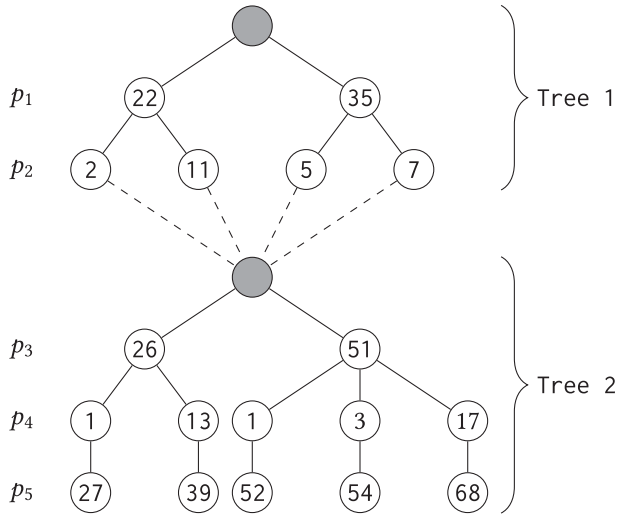

Fig. 1. The example chain-of-trees represents the search space of parameters $p_1, \ldots, p_5$.

To efficiently store constrained search spaces in memory, we introduce the novel *chain-of-trees* search space structure. This structure chains multiple trees, where each tree represents the search space part of a group of interdependent tuning parameters, as defined in the previous section.

We explain our chain-of-trees search space structure for a simple, illustrative example of five tuning parameters:

$$
\begin{array}{lll}
p_1 := (\, n_1, & \{22, 35\}, & \_ \quad), \\
p_2 := (\, n_2, & \{2, 5, 7, 11\}, & \texttt{divides}(n_1) \quad), \\
p_3 := (\, n_3, & \{26, 51\}, & \_ \quad), \\
p_4 := (\, n_4, & \{1, 3, 13, 17\}, & \texttt{divides}(n_3) \quad), \\
p_5 := (\, n_5, & \{27, 39, 52, 54, 68\}, & \texttt{equals}(n_3 + n_4) \quad).
\end{array}
$$

Here, for an easy distinction, the parameters' ranges comprise different values, and p_1 and p_3 have no constraints. We use `divides(N)` as an alias for the parameter constraint `[](int i) {return N % i == 0;}`, and we use `equals(N)` for the constraint `[](int i){return N == i;}`. There are two groups of interdependent parameters in the example; the first group comprises parameters $\{p_1, p_2\}$, while parameters $\{p_3, p_4, p_5\}$ form the second group.

Figure 1 illustrates our chain-of-trees structure for the example parameters $p_1, \ldots, p_5$. For each of the two parameter groups, we use a tree (`Tree 1` and `Tree 2`) to represent its part within the search space, and we chain these two trees by connecting the leaves of the first tree with the root of the second tree. To save memory, we store the connecting (dashed) edges as a single reference in `Tree 1`. Each path in `Tree 1` from the root to a leaf represents a valid configuration of parameters p_1 and p_2, for which their constraints are satisfied, and each combination of a path in `Tree 1` and a path in `Tree 2` represents a valid, full configuration.

In our chain-of-trees structure, parameter values are often stored only once, whereas in a plain array of configurations (as in CLTune and KernelTuner), these values would be repeated many times. For example, we store values 22 and 35 only once at the top level of `Tree 1` while these would be stored 20 times in the traditional space representation (once per configuration in the search space), resulting in a high memory footprint. Furthermore, the configurations of parame-

ters p_3, p_4, p_5, which are represented by `Tree 2`, would have to be stored for every leaf of `Tree 1` (four times in this example). We avoid this significant waste of memory by storing `Tree 2` only once and chaining the two trees together.

Note that our chain-of-trees structure is efficient also for storing the spaces of parameters without interdependencies: if tuning parameters are independent, then each single parameter represents its own interdependent parameter group (comprising only one parameter), which corresponds to exactly the same range-based search space representation as used in OpenTuner and `libtuning`.

5 EXPLORING CONSTRAINED SEARCH SPACES

The third and final phase of auto-tuning is the exploration of the search space, generated and stored as described in Sections 3 and 4, using some search technique. State-of-the-art general-purpose auto-tuning frameworks follow one of two basic approaches to the exploration phase.

CLTune and KernelTuner explore constrained search spaces, but they use a plain array of configurations. Consequently, they provide search techniques and only one-dimensional view on the search space, which often causes sub-optimal auto-tuning results, because locality information within the space's particular dimensions are lost [70]. For example, we demonstrate in Section 7 for the search by simulated annealing—CLTune's most efficient search technique [41]—that the selection time of the next candidate point is very high for large search spaces when relying on the one-dimensional space, thus leading to poor auto-tuning results.

OpenTuner and `libtuning` retain the multidimensionality of their search spaces, as required by search techniques for high search efficiency [70]. However, these two frameworks have to explore unconstrained search spaces which can contain also invalid configurations. This usually drastically worsens their efficiency for programs with interdependent tuning parameters, as we confirm experimentally in Section 7.

We aim at combining the advantages of both state-of-the-art approaches: we explore constrained search spaces (as in CLTune and KernelTuner), and we search in multiple dimensions (as in OpenTuner and `libtuning`). For this, we exploit the structure of our chain-of-trees search space representation introduced in the previous section.

As search techniques usually explore coordinate spaces (i.e., spaces containing equally sized sequences of real numbers that have no interdependencies among them), our basic idea for exploration is as follows: we map a coordinate space to our chain-of-trees representation; thereby, we reduce the challenge of exploring a chain-of-trees to the challenge of exploring a coordinate space—the most efficient structure for search techniques [70]. For a chain-of-trees with L levels (excluding the roots), we map to it a coordinate space with L dimensions. For each dimension in the coordinate space, we use real numbers in the interval $(0, 1]$—all points from 0 to 1, excluding 0. We denote the coordinate space as $(0, 1]^L$.

Figure 2 demonstrates an example of how we map a coordinate space to our chain-of-trees structure. For illustration, we use a chain-of-trees with four levels, i.e., $L = 4$ (roots excluded); correspondingly, we use a four-dimensional coordinate space for mapping it to the four-leveled chain-of-trees. The goal of our mapping is to assign to each arbitrary sequence $(l_1, \ldots, l_4) \in (0, 1]^4$ in the coordinate space a path in the chain-of-trees (and thus a configuration; see Section 4), which we do intuitively as follows. In level 1, the chain-of-trees has four nodes, so in the first dimension of the coordinate space, we split interval $(0, 1]$ evenly into four equally sized blocks, where each block corresponds to one node in the chain-of-tree's first level. In our example, we map each l_1 in block $(0, 0.25]$ to the root's first child ①, and if l_1 is in block $(0.25, 0.5]$, we map it to the root's second child ②, and so forth. In level 2, after moving along the path (s_1) which comprises only

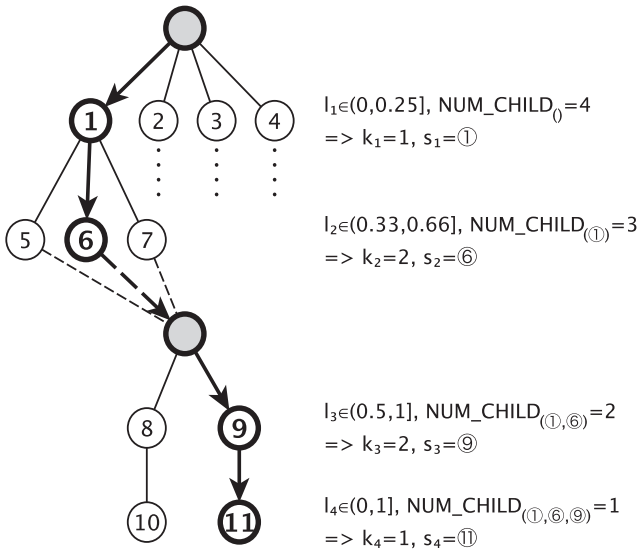

Fig. 2. Example of exploring our chain-of-trees structure in multiple dimensions, based on an L-dimensional coordinate space (in this example, $L = 4$). Subtrees in level 1—for nodes 2, 3, 4—are omitted for brevity.

node $s_1 = ①$, we have three nodes, so each $l_2 \in (0, 0.33]$ is mapped to s_1's first child node ⑤, and each $l_2 \in (0.33, 0.66]$ is mapped to the second child node ⑥, and so on.

In general, for an arbitrary L-leveled chain-of-trees, we map an L-dimensional coordinate space to this chain-of-trees as follows. Each sequence $(l_1, \ldots, l_L) \in (0, 1]^L$ in the coordinate space is mapped to a path $(s_1, \ldots, s_L)$ in the chain-of-trees. To obtain node s_i, $1 \leq i \leq L$, we calculate $k_i := \lceil l_i * \texttt{NUM\_CHILD}_{(s_1,\ldots,s_{i-1})} \rceil$, where $\texttt{NUM\_CHILD}_{(s_1,\ldots,s_{i-1})}$ is the number of child nodes of s_{i-1} after moving along the path $(s_1, \ldots, s_{i-1})$. We round up k_i to the next higher integer value (indicated by $\lceil \ldots \rceil$) and set s_i as the k_i-th child of node s_{i-1}.

6 USER INTERFACE OF ATF

While this article focuses on ATF's contributions in generating, storing, and exploring constrained search spaces, ATF has also a further major goal: simplifying the auto-tuning process to make it appealing to common application developers, which is the focus of previous work [49, 50]. For example, [49] shows that ATF's user interface simplifies the auto-tuning process for the user as compared to frameworks OpenTuner and CLTune: the ATF user annotates the program's source code with easy-to-use *tuning directives* which specify the tuning parameters (with their names, ranges, and possible constraints), the search technique, and the abort condition; ATF then automatically creates the corresponding special-purpose auto-tuner for generating, storing, and exploring the program-specific search space. In contrast, the users of OpenTuner and CLTune have to implement a low-level auto-tuning program in Python or C++, correspondingly.

Listing 3 shows how ATF is used for auto-tuning a popular example used in many auto-tuning case studies [13, 27, 34, 41, 53, 64, 66]—the OpenCL GEMM kernel of the CLBlast library [40], which computes general matrix-matrix multiplication on either CPU or GPU. The kernel, declared in lines 24 and 25, has 16 tuning parameters, e.g., the SIMD vector width `VWM` (line 5), the number of threads per thread group `MDIMC` (line 8), and the tile size `MWG` (line 12). The `atf::tp` directive specifies the tuning parameters for ATF with a name, range, and (optionally) a parameter constraint. The constraints express, e.g., that the local size `MDIMC` has to be less than or equal to the maximally

```
1  #atf::var::M         $1 // first  command line argument
2  #atf::var::N         $2 // second command line argument
3  #atf::var::K         $3 // third  command line argument
4
5  #atf::tp name        "VWM" // vector width
6           range       {1, 2, 4, 8}
7
8  #atf::tp name        "MDIMC" // local size
9           range       interval<size_t>( 1,M )
10          constraint  less_than_or_eq( atf::ocl::max_wi_size_0 )
11
12 #atf::tp name        "MWG" // tile size
13          range       interval<size_t>( 1,M )
14          constraint  multiple_of( MDIMC*VWM ) && ...
15
16 // 13 further tuning parameter
17
18 #atf::search_technique simulated_annealing
19 #atf::abort_condition  evaluations( 10000 )
20
21 // ... OpenCL-specific directives
22
23 // CLBlast's OpenCL GEMM kernel
24 __kernel void XgemmDirectTN( ... )
25 { ... }
```

Listing 3. ATF directives for auto-tuning CLBlast's GEMM routine (some parameters omitted for brevity).

hardware-supported thread size `atf::ocl::max_wi_size_0` (line 10), and that the tile size `MWG` has to be a multiple of the local size `MDIMC` multiplied with the vector width `VWM` (line 14), which is an interdependency among these three parameters. Here, `less_than_or_eq(x)` and `multiple_of(x)` are so-called *constraint aliases* which ATF provides for user's convenience; they are automatically replaced by ATF with the constraint functions `[](auto i) {return i < = x;}` and `[](auto i){ return i % x == 0;}`, respectively. The two functions take as input an element `i` of parameter's range, and they return either `true` or `false`, indicating whether the constraint is satisfied or unsatisfied for range element `i`. The `auto` specifier represents a generic type in ATF, allowing the user to use constraint aliases for differently typed tuning parameters. ATF provides several constraint aliases, and it allows the user to specify arbitrary, self-defined constraints as unnamed C++17 functions (lambda expressions).

Further in Listing 3, the search technique and abort condition, i.e., when to stop the auto-tuning process, are specified in lines 18 and 19. ATF provides special directives for OpenCL programmers (omitted for brevity in the listing) to automatically generate the host code in which the user can set, e.g., the target device and kernel's input parameters (line 21). ATF provides various search techniques, e.g., simulated annealing and AUC bandit which combines multiple techniques for exploration (such as differential evolution, Nelder-Mead, and Torczon hillclimbers) [49]. ATF also offers further tuning directives, e.g., for auto-tuning programs written in arbitrary programming languages and for arbitrary tuning objectives (e.g., high runtime performance and/or low energy consumption). We do not discuss ATF's supported search techniques and its further directives, because this is the focus of previous work [49, 50].

7 EXPERIMENTAL EVALUATION

All experiments described in this section can be reproduced using our artifact implementation [4].

In the following, after describing our experimental setup in Section 7.1, we introduce four application case studies in Section 7.2 which we use for experiments in this section. Afterwards, in Section 7.3, we compare ATF which implements our novel mechanisms for search space generation (Section 3), storing (Section 4), and exploration (Section 5)—the three main contributions of this article—against the state-of-the-art competitors. In Section 7.4, we experimentally analyze

Table 1. Auto-Tuning Characteristics of Our Four Application Studies

	App.	Input Size	#TPs	TP Groups	Min. RS	Max. RS	Med. RS	Avg. RS	|SP|	FT
1	CONV	$4,096 \times 4,096$	14	{1, 1, 2, 2, 4, 4}	2	4,092	4,092	2,339.29	$1.69 * 10^8$	$1.49 * 10^{-23}$
2	GEMM	$10 \times 500 \times 64$	19	{1, 1, 2, 3, 4, 4, 4}	2	500	10	121.84	$2.51 * 10^8$	$2.47 * 10^{-17}$
3	CCSD(T)	$24 \times 16 \times 16 \times 24 \times 16 \times 16 \times 24$	39	{1, 1, 2, 4, 4, 4, 4, 4, 4, 7}	2	24	16	15.46	$8.81 * 10^{18}$	$1.47 * 10^{-25}$
4	PRL	$1,024 \times 1,024$	14	{1, 1, 2, 2, 4, 4}	2	1,024	1,024	586.14	$2.31 * 10^7$	$1.33 * 10^{-19}$

ATF regarding each particular phase of the auto-tuning process. Finally, we discuss in Section 7.5 ATF's auto-tuning efficiency for further application classes, and we present in Section 7.6 ATF's efficiency for a real-world deep learning application.

7.1 Experimental Setup

We use a system equipped with an Intel Xeon Gold 6140 18-core CPU, tacted at 2.3 GHz with 192 GB main memory and hyper-threading enabled, and a NVIDIA Tesla V100-SXM2-16GB GPU. Time measurements are made using the C++ chrono library and the OpenCL profiling API, respectively.

7.2 Application Case Studies for Experiments

Our experiments rely on four case studies: (1) Gaussian Convolution (CONV), which is a popular stencil computation, (2) GEneral Matrix-Matrix multiplication (GEMM), which is a linear algebra routine, (3) Coupled Cluster (CCSD(T)), which is important in quantum chemistry [14], and (4) Probabilistic Record Linkage (PRL), which is widely used in data mining [52]. We use the recent CPU and GPU implementations of these four applications presented in [51], and we show experimentally that when using ATF for auto-tuning, these implementations can be auto-tuned to better performance than current state-of-practice solutions, e.g., Facebook's TensorComprehensions library [67], which relies on state-of-the-art polyhedral techniques combined with a special-purpose auto-tuner, as well as hand-optimized vendor libraries such as Intel MKL/MKL-DNN [21, 22] and NVIDIA cuBLAS/cuDNN [42, 44] for linear algebra routines and stencil computations, respectively.

The implementations in [51] are written in OpenCL in order to target different kinds of architectures, and they rely on multiple tuning parameters, including sizes of tiles and numbers of threads on different memory and core layers. The parameters have various interdependencies among them, e.g., the value of a tile size tuning parameter on an upper memory layer has to be a multiple of a tile on a lower memory layer—an interdependency among the tile size parameters–because a lower-layer tile is a chunk of an upper-layer tile. We refer the reader to [51] for more details about the tuning parameters and their corresponding interdependencies, as this is not the focus of this article.

Relevant for the evaluation in this article are the auto-tuning characteristics of our four studies, which are summarized in Table 1: (1) number of tuning parameters (denoted as *#TPs* in the table); (2) number of groups of interdependent tuning parameters, as defined in Section 3, and the groups' individual sizes (*TP Groups*); (3) minimum parameter range size (*Min. RS*); (4) maximum parameter range size (*Max. RS*); (5) median parameter range size (*Med. RS*); (6) average parameter range size (*Avg. RS*); (8) size of the constrained search space ($|SP|$); and (9) fraction (*FT*) of the unconstrained search space that represents valid configurations. For example, application CONV has 14 tuning parameters which are automatically split by ATF into six groups of interdependent

parameters: two groups comprising one parameter, two groups comprising two parameters, and two groups comprising four parameters. Characteristics *Min. RS*, *Max. RS*, *Med. RS*, *Avg. RS*, $|SP|$, and *FT* depend on the particular input size of the applications, because the input size is used to calculate the upper bound of some of the tuning parameters' ranges. For example, the range of the tile size tuning parameter is defined as all values between 1 and the input size [51]. In the table, we present values of the input-dependent characteristics for the same input sizes as also used in [51], e.g., real-world sizes taken from deep learning. For example, for input size $4{,}096 \times 4{,}096$, `CONV`'s parameters have a minimum range size of 2 (a Boolean parameter) and a maximum range size of 4,092 (tile size); `CONV`'s median range size is 4,092, and on average, the ranges of `CONV`'s tuning parameters contain 2,339.29 values. The constrained search space of `CONV` (which comprises only valid configurations, as in ATF, CLTune, and KernelTuner) has a size of $1.69 * 10^8$ for input size $4{,}096 \times 4{,}096$. The `FT` value ($1.49 * 10^{-23}$ in this example) denotes which fraction of the unconstrained search space represents valid configurations. For example, the unconstrained search space of `CONV` has a size of $1.13 * 10^{31}$ and only a fraction of $1.49 * 10^{-23}$ of configurations within the space ($= 1.69 * 10^8$ many) are valid.

Table 1 shows that our case studies have very different auto-tuning characteristics, thus enabling a thorough evaluation of the ATF framework.

7.3 Comparison of Auto-Tuning Efficiency

We compare the auto-tuning efficiency of ATF which implements our novel mechanisms presented in Sections 3–5 to the auto-tuning efficiency of (i) OpenTuner which is designed for programs whose tuning parameters have no interdependencies; (ii) CLTune which supports interdependencies among tuning parameters; and (iii) ATF's former implementation [49, 50] which relies on prototype mechanisms for search space generation, storing, and exploration. For brevity, we do not present our experimental results for the general-purpose auto-tuning frameworks `libtuning` and `KernelTuner`, because our results for them are analogous to those that we obtain for OpenTuner and CLTune: the same as OpenTuner, `libtuning` is optimized toward programs whose tuning parameters have no interdependencies, and KernelTuner relies on exactly the same mechanisms for search space generation and storing as CLTune. Thus, even though `libtuning` and `KernelTuner` use other kinds of search techniques than OpenTuner and CLTune, both have difficulties with auto-tuning our case studies for the same reasons as OpenTuner and CLTune.

Additionally, we compare the performance of our application case studies auto-tuned using ATF against the newest versions of state-of-practice, high-performance computing libraries that use their own optimized implementations of these applications: (a) Intel `MKL-DNN` 0.21.5/`MKL` 2020 [21, 22] and NVIDIA `cuDNN` 7.6.5/`cuBLAS` 10.2 [42, 44] which are architecture-specific approaches for high-performance convolution computations and linear algebra routines on CPU and GPU, respectively; the libraries rely on hand-optimized assembly code, rather than auto-tuned OpenCL programs as we do; (b) `Conv2D` and `CLBlast` [40, 41] which are auto-tunable OpenCL implementations of convolution and matrix multiplication, respectively, for CPUs and GPUs; both implementations rely on CLTune for auto-tuning, which is specifically designed and optimized toward auto-tuning these two implementations [41]; (c) `TensorComprehensions` [67] and `COGENT` [29] which are recent approaches optimized toward efficiently computing `CCSD(T)` on NVIDIA GPUs; `TensorComprehensions` generates its own `CCSD(T)` implementation based on state-of-the-art polyhedral techniques, and it is tightly coupled to a self-provided, special-purpose auto-tuner; COGENT generates CUDA code for `CCSD(T)` that relies on hand-crafted heuristics for optimization, rather than auto-tuning; and (d) the hand-optimized, parallel Java implementation of `PRL` for multi-core CPU that is currently used by *EKR* [20]—the largest cancer registry in Europe.

Table 2. Auto-Tuning Efficiency of ATF vs. State-of-the-Art Auto-Tuners and High-Performance Libraries on CPU (Left Part of the Table) and GPU (Right Part) for Application Studies: CONV (Gaussian Convolution), GEMM (GEneral Matrix-Matrix Multiplication), CCSD(T) (Coupled Cluster), and PRL (Probabilistic Record Linkage)

		Intel Xeon Gold 6140 CPU					NVIDIA V100 GPU				
		SP Gen.Time	SP Size	Valid Configs.	Invalid Configs.	Runtime on CPU	SP Gen.Time	SP Size	Valid Configs.	Invalid Configs.	Runtime on GPU
CONV	**ATF**	63 ms	1.69E+08	15,804	0	**4.35 ms**	68 ms	1.69E+08	5,178	0	**0.20 ms**
	OpenTuner [2]	--	1.13E+31	0	182,918	**FAILED**	--	1.13E+31	0	161,355	**FAILED**
	CLTune [41]	>4.0 h	1.69E+08	0	0	**FAILED**	>4.0 h	1.69E+08	0	0	**FAILED**
	CLTune (pruned) [41]	1227 ms	48	48	0	**194.14 ms**	1481 ms	48	48	0	**5400.07 ms**
	ATF (former) [49,50]	>4.0 h	1.69E+08	0	0	**FAILED**	>4.0 h	1.69E+08	0	0	**FAILED**
	MKL-DNN [22] / cuDNN [44]	*auto-tuning not supported*				***13.49 ms***	*auto-tuning not supported*				***3.14 ms***
	Conv2D [41]	*3358 ms*	*3,536*	*3,536*	*0*	***16.20 ms***	*3986 ms*	*3,684*	*3,684*	*0*	***0.80 ms***
GEMM	**ATF**	5 ms	2.51E+08	22,463	0	**26.74 us**	4 ms	2.51E+08	7,426	0	**5.12 us**
	OpenTuner [2]	--	1.02E+25	0	178,008	**FAILED**	--	1.02E+25	0	158,066	**FAILED**
	CLTune [41]	>4.0 h	2.51E+08	0	0	**FAILED**	>4.0 h	2.51E+08	0	0	**FAILED**
	CLTune (pruned) [41]	1.3 h	3,024	2,142	0	**58.92 us**	1.6 h	3,024	2,300	0	**613.99 us**
	ATF (former) [49,50]	34 min	2.51E+08	9,837	0	**30.06 us**	25 min	2.51E+08	6,495	0	**5.32 us**
	MKL [21] / cuBLAS [42]	*auto-tuning not supported*				***65.03 us***	*auto-tuning not supported*				***12.29 us***
	CLBlast [40]	*107 ms*	*3839*	*3839*	*0*	***160.00 us***	*134 ms*	*8,420*	*8,420*	*0*	***23.00 us***
CCSD(T)	**ATF**	12 ms	8.81E+18	7,702	0	**3.67 ms**	11 ms	8.81E+18	2,270	0	**0.23 ms**
	OpenTuner [2]	--	5.99E+43	0	190,651	**FAILED**	--	5.99E+43	0	168,090	**FAILED**
	CLTune [41]	>4.0 h	8.81E+18	0	0	**FAILED**	>4.0 h	8.81E+18	0	0	**FAILED**
	CLTune (pruned) [41]	>4.0 h	2.32E+09	0	0	**FAILED**	>4.0 h	2.32E+09	0	0	**FAILED**
	ATF (former) [49,50]	57 ms	8.81E+18	5,859	0	**5.95 ms**	93 ms	8.81E+18	2,439	0	**0.24 ms**
	TensorComprehensions [67]	*cpu not supported*					--	*?*	*3,190*	*0*	***0.54 ms***
	COGENT [29]	*cpu not supported*					*auto-tuning not supported*				***0.48 ms***
PRL	**ATF**	17 ms	2.31E+07	291	0	**0.41 ms**	18 ms	2.31E+07	105	0	**0.74 ms**
	OpenTuner [2]	--	1.74E+26	0	188,030	**FAILED**	--	1.74E+26	0	163,849	**FAILED**
	CLTune [41]	>4.0 h	2.31E+07	0	0	**FAILED**	>4.0 h	2.31E+07	0	0	**FAILED**
	CLTune (pruned) [41]	1450 ms	1.75E+05	442	0	**0.68 ms**	1818 ms	1.75E+05	162	0	**0.79 ms**
	ATF (former) [49,50]	>4.0 h	2.31E+07	0	0	**FAILED**	>4.0 h	2.31E+07	0	0	**FAILED**
	EKR [20]	*auto-tuning not supported*				***183.00 ms***	*gpu not supported*				

Table 2 shows the measured runtimes of our four case studies on CPU (left part of the table) and GPU (right part) when auto-tuned using ATF, compared to auto-tuning the application studies using the existing general-purpose auto-tuners listed above. We auto-tune each of our four studies for 4 h with each auto-tuning framework; the studies are denoted in the table as ATF, OpenTuner, CLTune, and CLTune (pruned) which is CLTune with an expert-pruned search space, and ATF (former) which is ATF's prototype implementation [49, 50]. For a fair comparison, we conduct each tuning run 10 times, and we present for each framework the results of the best run. High-performance libraries that rely on their own implementations of our application studies are also presented in the table and highlighted in italic.

In addition to the runtimes of the application studies, we present in Table 2 for each framework and library also (i) the time required for generating a study's search space, (ii) the search space size for each particular study, (iii) the number of valid configurations explored in the 4h of tuning time for each study, and (iv) the number of invalid configurations explored. Note that ATF, CLTune, and ATF (former) explore only valid configurations, but at the cost of the search space generation time. In contrast, OpenTuner relies on the unconstrained search spaces and

thus, it requires no search space generation time; however, at the cost of invalid configurations within its search space. The high-performance libraries Intel MKL-DNN/MKL and NVIDIA cuDNN/cuBLAS (for CONV and GEMM), as well as libraries COGENT (for study CCSD(T)) and EKR (for PRL) do not rely on auto-tuning; therefore, they do not generate or explore search spaces. The TensorComprehensions library uses internally a domain-specific, special-purpose auto-tuner which explores valid configurations only. To make comparison more challenging for ATF, we use for TensorComprehensions a tuning time of 12h, rather than 4h as for ATF. Note that we cannot report the search space size of TensorComprehensions, because the size is not listed in its log files.

We observe in Table 2 that frameworks OpenTuner, CLTune, and ATF (former) have difficulties with auto-tuning most of our application studies. In the case of CLTune and ATF (former), this is because they require a too high search space generation time, which we discuss and analyze in detail in the next subsection. OpenTuner cannot find a single valid configuration in the tuning time of 4h, in all 10 tuning runs per particular application, because it relies on the unconstrained search space which contains too many invalid configurations.

For CLTune in Table 2, we use also a hand-pruned search space (denoted as CLTune (pruned) in the table), because pruning is usually required in CLTune for generating its search spaces in adequate time [41]. To generate the pruned CLTune search spaces, we use exactly the same restricted ranges of tuning parameters that are recommended by the CLTune experts [41]: range $\{1, 8, 16, 32\}$ for the number of threads on different layers, rather than the parameters' complete range $\{1, \ldots, N\}$ which we use for ATF, where N is the input size; for sizes of tiles, we use range $\{1, 16, 32, 64, 128\}$ instead of $\{1, \ldots, N\}$. For the further tuning parameters of our four studies, the CLTune experts provide no pruning recommendations; thus, to avoid missing well-performing values in these parameters' ranges, we use for them the original, unrestricted ranges which we also use for ATF. We observe in Table 2 that search space pruning enables CLTune to generate its search space in acceptable time. For example, for application CONV, CLTune (pruned) needs 1.227s to generate CONV's search space, while without pruning, CLTune needs >4h. However, pruning severely affects applications' performance: slowdowns ranging from 1.06× (for PRL) to up to $>10^4$× (for CONV) when comparing CLTune (pruned) to ATF on GPU. This is because pruning by hand is a complex task and thus, it often excludes well-performing configurations out of the search space, even when using the pruned parameter ranges recommended by the CLTune experts. For example, for application CONV on GPU, ATF determines as optimal number of threads the (counter-intuitive [43]) value of 372, which is not represented in CLTune's recommended, hand-pruned ranges. ATF is able to find such counter-intuitive parameter values, because classification of configurations in well-performing and not well-performing is left entirely to ATF (without relying on the programmer for hand-pruning).

As compared to high-performance libraries, we observe in Table 2 that ATF auto-tunes our applications to better performance. In the case of MKL-DNN, MKL, cuDNN, cuBLAS, COGENT, and EKR, this is because we rely on auto-tuning for the particular input size, while the libraries use hand-crafted heuristics optimized toward average high performance over different sizes; thereby, the libraries avoid the time-intensive process of auto-tuning for the particular size. However, as we demonstrate in Section 7.6, auto-tuning for the input size is well amortized in many application areas [63], e.g., deep learning, where the same sizes are reused in each program run.

High-performance libraries Conv2D and CLBlast in Table 2 provide their own, auto-tunable OpenCL implementations that rely on CLTune for auto-tuning. In contrast to these two libraries, we achieve better performance by auto-tuning with ATF the implementations provided in [51]; these implementations have larger search spaces than Conv2D and CLBlast and thus, they enable a more fine-grained optimization for the target architecture and input/output characteristics (this is discussed in detail in [51]). The larger spaces in [51] cannot be generated using CLTune (as

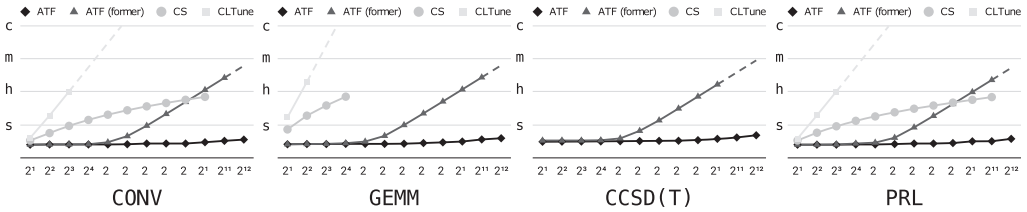

Fig. 3. Search space generation time (lower is better) of ATF vs. ATF's former implementation [49, 50], Constraint Solver (**CS**) [39], and CLTune [41] for our four case studies using different square, power-of-two input sizes. We use a logarithmic scale on the y-axis: seconds (s), hours (h), months (m), centuries (c). When search space generation time exceeds 12 h, we use a theoretically computed generation time (highlighted as a dashed line). For study `CCSD(T)`, we cannot present the search space generation time of `CLTune` and `CS`, because `CLTune` requires >12h generation time for each input size of `CCSD(T)`, and `CS` crashes due to large memory footprint. `CS` also crashes for other applications on large sizes because of its large memory footprint.

confirmed in Table 2), because `CLTune` relies on a straightforward search space generation process (discussed in Section 3).

In the following, we compare ATF to the state-of-the-art frameworks in terms of each particular phase of the auto-tuning process.

7.4 Generating, Storing, and Exploring Constrained Search Spaces

This section experimentally evaluates ATF's three main contributions presented in this article (Sections 3–5) by measuring and assessing the search space generation time, the memory footprint, and the exploration efficiency of ATF for constrained search spaces as compared to CLTune and ATF's prototype implementation [49, 50]. In particular, we show that even when improving search space generation in the state-of-the-art auto-tuning frameworks, which is one of their main limitations (as discussed in the previous subsection), the approaches would still suffer from severe weaknesses regarding search space storing and exploration.

Note that a comparison with OpenTuner and `libtuning` for generating/storing/exploring constrained search spaces is not possible, because both approaches rely on the unconstrained search space (see Section 2.1), with the major drawbacks discussed in Section 7.2. We also refrain from presenting our experimental results for KernelTuner, because it relies on exactly the same mechanisms for generating, storing, and exploring constrained search spaces as CLTune; thus, both approaches achieve analogous results, even though KernelTuner uses other kinds of search techniques.

Generating Constrained Search Spaces. Figure 3 reports the measured search space generation time for our four application case studies when using ATF which implements our novel mechanism for generating constrained search spaces (introduced in Section 3), compared to the search space generation mechanisms of CLTune and ATF's former implementation [49, 50]. In addition, we compare also to a state-of-the-art constraint solver [39] which is designed for combinatorial problems and thus can be exploited also for search space generation in auto-tuning. We show for each application the generation time for different square, power-of-two input sizes. The generation times are growing with the input sizes, because the sizes are used to calculate the upper bounds for some of the tuning parameter ranges, e.g., tile size parameters. For each combination of application and input size, we measure the search space generation times up to 12 h on our system. For larger input sizes, the generation times of our competitors often exceed 12 h (e.g., in the case of study `CONV`, for sizes $2^4, 2^5, \ldots$). In these cases, we report a theoretically computed generation time

(highlighted as dashed lines in Figure 3) which we compute based on the average times measured for smaller input sizes.

We observe in Figure 3 that ATF generates constrained search spaces faster than its competitors, by several orders of magnitude, already on small input sizes (note the logarithmic scale in the figure). For example, CLTune requires > 1h for generating the search space of CONV even for small 8×8 input images, while ATF requires only 21ms to generate the same space—a speedup of $> 10^5$. This is because CLTune relies on a naive search space generation algorithm (discussed in Section 3) which iterates over the huge unconstrained search space containing more than 10^9 configurations for 8×8 input images. In contrast, ATF uses our novel search space generation algorithm which enables generating groups of interdependent parameters independently and in parallel (referred to as *Optimization 1* in Section 3), and ATF also checks constraints early in the search space generation process (*Optimization 2*).

Compared to ATF's prototype implementation [49, 50], our new search space generation algorithm exploits parameter constraints for each particular parameter, while the former implementation of ATF used a proof-of-concept search space generation algorithm in which, for simplicity, parameter constraints were checked for all parameters within a group at the group's last parameter. This is sufficient for ATF's initially targeted application class—BLAS routines on small input sizes—and makes implementation in [49, 50] simpler, however, at the cost of a high search space generation time for other important applications, as shown in Table 2.

In contrast to ATF's former implementation, the constraint solver (CS in Figure 3) uses parameter constraints for each particular parameter. However, the solver does not exploit parameter grouping and parallelization (ATF's *Optimization 1*), causing significantly higher search space generation time than our novel space generation algorithm in ATF.

Asymptotic behavior differs over approaches, because ATF and the solver check constraints early for each particular parameter, while ATF's former implementation and CLTune check constraints late, for entire groups of parameters (ATF former) or simultaneously for all parameters (CLTune).

Storing Constrained Search Spaces. We compare the memory requirements of the constrained search space built by ATF which relies on its novel chain-of-trees structure (discussed in Section 4) against CLTune and ATF's former implementation for our four case studies using again different square, power-of-two input sizes. CLTune and ATF's former implementation rely on the same, memory-intensive, one-dimensional search space representation. Consequently, both approaches suffer from a memory crash for already quite small input sizes.

In the following, to make comparison challenging for ATF, we compute theoretically the minimum memory requirement of the one-dimensional search space representation in CLTune and ATF's former implementation for each particular combination of application and input size, as follows. In both frameworks, the search space is a flat array of configurations which comprise particular values of the tuning parameters (e.g., 14 parameters in the case of application CONV; see column #TPs in Table 1). Each tuning parameter value has at least a size of 1 `byte` (usually more, e.g., 4 `byte` in the case of an `integer` parameter); this results in the following minimum memory requirement of the one-dimensional search space: `|SP| * #TPs * 1 Byte`. Here, |SP| denotes the number of configurations within the space, which is equal to the search space size; #TPs denotes the number of parameter values per configuration, which is equal to the number of tuning parameters. We compute |SP| for all applications and input sizes using ATF which (in contrast to CLTune and former ATF) is capable of generating and storing large spaces.

In Figure 4, we observe for all four applications that our chain-of-trees search space representation in ATF requires significantly less memory than the one-dimensional space representation

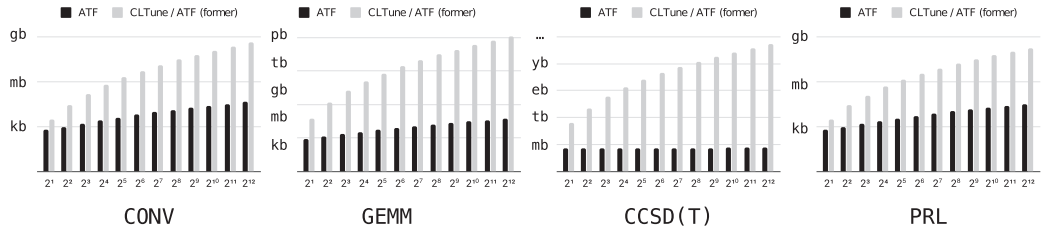

Fig. 4. Memory footprint (lower is better) of ATF vs. CLTune [41] and ATF's former implementation [49, 50] (both rely on same search space representation) for our four application case studies using different square, power-of-two input sizes. We use a logarithmic scale on the y-axis.

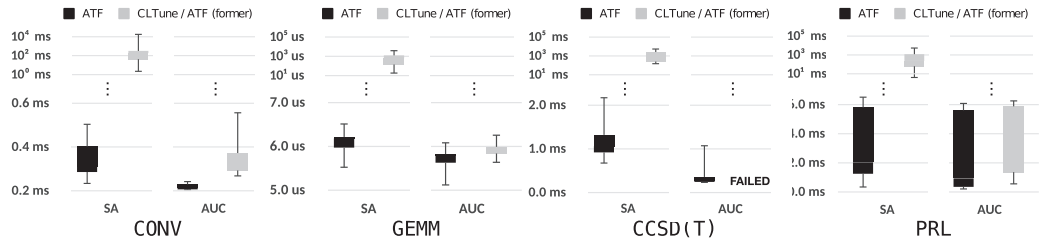

Fig. 5. Search space exploration efficiency (lower is better) of ATF vs CLTune [41] and ATF's former implementation [49, 50] (both rely on same search space representation) for our four application case studies using search techniques Simulated Annealing (SA) and AUC bandit (AUC). We use a logarithmic scale on the y-axis.

used in CLTune and former ATF (note the logarithmic scale in the figure). For example, application `CCSD(T)`'s one-dimensional search space requires for input size 2^{12} the prohibitively high amount of $>10^{19}$ GB (calculated according to the formula above), while ATF's search space structure requires only 256KB for storing the same space—a memory consumption reduced by a factor of $>10^{22}$.

Exploring Constrained Search Spaces. Figure 5 shows the advantages of ATF's multi-dimensional exploration strategy (described in Section 5) over a one-dimensional strategy (as in CLTune and ATF's prototype implementation) drawn as box plots. Each plot shows 10 runtimes of a particular study, obtained after 10 independent auto-tuning runs of 4 h each. A box depicts the 25%–75% quartiles, i.e., half of the configurations obtained after the 10 independent auto-tuning runs achieve a runtime that lays within the box. The vertical lines connect for each study the runtime of the worst auto-tuning run (i.e., the final configuration after 10 runs that achieves the least runtime) with the runtime of the best run. We auto-tune all applications for NVIDIA Tesla V100 GPU using the input sizes from Table 1. Note that different tuning runs usually find different final configurations, because search techniques are not deterministic; for example, techniques usually start exploration at a random configuration within the search space.

Figure 5 confirms that, when exploring a constrained search space in multiple dimensions (as described in Section 5), by exploiting the structure of ATF's chain-of-trees space representation, we usually find better-performing parameter configurations in the same auto-tuning time (4 h in the figure) as compared to the traditional, one-dimensional exploration strategy in CLTune and ATF's former implementation.

For the simulated annealing search—CLTune's most efficient search technique [41]—we observe considerably better tuning results for the multi-dimensional exploration strategy in ATF, compared

to exploration in only one dimension: an average speedup of $10^2\times$, i.e., we can improve the runtime of our case studies on average by $10^2\times$ when using ATF for 4 h of auto-tuning as compared to auto-tuning the applications for 4 h using CLTune or the former ATF. This is because simulated annealing requires a long time for selecting the next configuration when the search space is one-dimensional and large [10]. When exploiting ATF's chain-of-trees structure, we perform exploration in multiple dimensions; in each dimension, we explore a corresponding level of the chain-of-trees structure, rather than in the large search space. This allows more configurations to be explored and results in better configurations being found in 4 h of tuning time.

We also observe in Figure 5 better exploration efficiency for the AUC bandit search—currently one of the most efficient search techniques [2, 49]—when relying on ATF's multi-dimensional exploration approach. This is because exploration in multiple dimensions enables better exploiting locality information within space's particular dimensions [70], which is especially beneficial for large spaces, e.g., the space of `CCSD(T)`.

Note that our chain-of-trees search space structure enables at least the same exploration efficiency as the traditional exploration strategy in one dimension, for any search technique: search techniques can straightforwardly access our chain-of-trees search space structure in a one-dimensional fashion (exactly as in CLTune and ATF's former implementation) by iterating over the chain-of-tree's leaves.

7.5 ATF for Further Application Classes

ATF has been already successfully used for auto-tuning applications from different important domains [18, 31, 40, 48, 49, 51, 52, 58, 59]. For example, ATF (already in its prototype implementation [49, 50]) has proved to achieve tuning results of the same or even higher quality as OpenTuner and CLTune for their favorable application classes [49]: (i) GCC compiler's optimization flags [2] (favorable for OpenTuner), as an example application whose tuning parameters have no interdependencies, and (ii) CLBlast library's GEMM implementation [40] (favorable for CLTune) whose tuning parameters have interdependencies among them.

Compared to OpenTuner, this is because GCC flags' tuning parameters have no interdependencies, causing both OpenTuner and ATF to generate, store, and explore the same (unconstrained) search space, and because ATF provides (among others) the highly efficient AUC bandit search technique [49] which is also used by OpenTuner for search space exploration. Consequently, both OpenTuner and ATF achieve for GCC flags the same good auto-tuning results.

As compared to CLTune, ATF is able to auto-tune the CLBlast's GEMM routine to better performance than CLTune, by up to 17× (as shown in [49]), even though CLTune is specifically designed toward auto-tuning this routine [41]. This is because the CLTune user has to massively hand-prune the ranges of GEMM's tuning parameters (i.e., remove valid values out of the ranges) in order to generate CLTune's search space in acceptable time. However, such pruning massively shrinks GEMM's search space, by factors $>1000\times$, thereby usually losing well-performing configurations within the space [49].

7.6 ATF for a Real-World Application

ATF can significantly speed up real-world applications that rely on compute-intensive kernels like those discussed in Section 7.3. We demonstrate this for the real-world example `siamese` which is used for handwriting recognition within the popular deep-learning framework Caffe [25].

Table 3 shows our performance analysis for `siamese`. We observe that GEMM is called in `siamese` over 50 million times in total, on 25 input sizes (which remain fixed for different inputs of `siamese` [25]). For computing GEMM on CPU, the `siamese` implementation in Caffe relies on the

Table 3. Auto-Tuning Efficiency of ATF for the siamese Neural Network (Runtimes in ms)

No.	Input Size			num calls	Intel Xeon Gold 6140 CPU					NVIDIA V100 GPU				
					ATLAS	CLBlast		ATF		cuBLAS	CLBlast		ATF	
	M	N	K		runtime	runtime	speedup over ATLAS	runtime	speedup over ATLAS	runtime	runtime	speedup over cuBLAS	runtime	speedup over cuBLAS
1.	50	64	500	8420128	0.2760	0.4720	***0.58***	0.0366	***7.55***	0.0133	0.0331	***0.40***	0.0061	***2.17***
2.	20	576	25	8420128	0.0848	0.1165	***0.73***	0.0281	***3.01***	0.0123	0.0213	***0.58***	0.0041	***3.00***
3.	20	576	1	8420128	0.0263	0.0539	***0.49***	0.0102	***2.57***	0.0061	0.0198	***0.31***	0.0041	***1.50***
4.	50	64	1	8420128	0.0038	0.0160	***0.24***	0.0025	***1.54***	0.0072	0.0199	***0.36***	0.0041	***1.75***
5.	500	64	50	6400000	0.2705	0.2147	***1.26***	0.0224	***12.06***	0.0082	0.0225	***0.36***	0.0061	***1.33***
6.	50	500	64	6400000	0.2668	0.2539	***1.05***	0.0455	***5.86***	0.0225	0.0224	***1.00***	0.0061	***3.67***
7.	20	25	576	6400000	0.0612	0.5655	***0.11***	0.0269	***2.28***	0.0143	0.0345	***0.42***	0.0061	***2.33***
8.	64	500	800	100002	1.1326	2.3305	***0.49***	0.1581	***7.17***	0.0195	0.0440	***0.44***	0.0195	***1.00***
9.	64	10	500	100002	0.0724	0.4678	***0.15***	0.0224	***3.23***	0.0133	0.0329	***0.40***	0.0061	***2.17***
10.	64	500	1	100002	0.0419	0.0652	***0.64***	0.0157	***2.67***	0.0061	0.0198	***0.31***	0.0041	***1.50***
11.	64	10	1	100002	0.0047	0.0109	***0.43***	0.0012	***4.03***	0.0061	0.0195	***0.31***	0.0041	***1.50***
12.	64	2	10	100002	0.0031	0.0176	***0.18***	0.0008	***3.91***	0.0123	0.0200	***0.62***	0.0041	***3.00***
13.	64	2	1	100002	0.0015	0.0095	***0.16***	0.0012	***1.25***	0.0072	0.0193	***0.37***	0.0041	***1.75***
14.	500	800	64	100000	1.0922	1.2813	***0.85***	0.0863	***12.65***	0.0123	0.0296	***0.41***	0.0164	***0.75***
15.	64	800	500	100000	1.1098	1.9104	***0.58***	0.0850	***13.06***	0.0174	0.0375	***0.46***	0.0215	***0.81***
16.	64	500	10	100000	0.0749	0.0889	***0.84***	0.0227	***3.30***	0.0133	0.0212	***0.63***	0.0041	***3.25***
17.	10	500	64	100000	0.0686	0.1517	***0.45***	0.0312	***2.20***	0.0184	0.0215	***0.86***	0.0051	***3.60***
18.	64	10	2	100000	0.0053	0.0122	***0.43***	0.0017	***3.14***	0.0072	0.0199	***0.36***	0.0041	***1.75***
19.	2	10	64	100000	0.0036	0.0218	***0.16***	0.0013	***2.66***	0.0174	0.0215	***0.81***	0.0041	***4.25***
20.	100	500	800	20200	1.5435	2.3467	***0.66***	0.2584	***5.97***	0.0256	0.0500	***0.51***	0.0256	***1.00***
21.	100	10	500	20200	0.0753	0.4875	***0.15***	0.0256	***2.95***	0.0133	0.0325	***0.41***	0.0061	***2.17***
22.	100	500	1	20200	0.0629	0.0687	***0.92***	0.0173	***3.64***	0.0061	0.0203	***0.30***	0.0041	***1.50***
23.	100	10	1	20200	0.0073	0.0141	***0.52***	0.0017	***4.21***	0.0072	0.0197	***0.36***	0.0041	***1.75***
24.	100	2	10	20200	0.0049	0.0279	***0.18***	0.0017	***2.87***	0.0113	0.0200	***0.56***	0.0041	***2.75***
25.	100	2	1	20200	0.0025	0.0125	***0.20***	0.0011	***2.18***	0.0072	0.0201	***0.36***	0.0041	***1.75***

ATLAS library [69] which uses a self-provided special-purpose auto-tuner for optimization; for GEMM on GPU, siamese uses the hand-optimized NVIDIA cuBLAS library which we discussed in Section 7.3. Alternatively to ATLAS and cuBLAS, the Caffe user can optionally choose the CLBlast library (also discussed in Section 7.3), which relies on CLTune for auto-tuning.

Table 3 shows that the siamese application requires for computing GEMM on CPU via ATLAS in total 2.1 h, which makes up 53% of siamese's total runtime on CPU (3.9 h); on GPU, siamese requires 10 min for GEMM via cuBLAS, which is 83% of siamese's total GPU runtime (13 min). However, when replacing ATLAS and cuBLAS by the ATF-optimized GEMM implementation in [51] (which we discussed in Section 7.3), we can speed up siamese's total runtime by 1.78× on CPU (to 2.2 h) and by 1.85× on GPU (to 7 min). This is because ATF is capable of auto-tuning the GEMM implementaiton in [51] to higher performance than ATLAS and cuBLAS, by up to 13× on CPU and 4× on GPU, as shown in Table 3.

ATF achieves better performance than ATLAS, because ATLAS relies on small search spaces and ignores correlations among tuning parameters by auto-tuning parameters independently of each other; most likely, this is done in ATLAS to simplify the auto-tuning process which would otherwise require similar mechanisms as presented for ATF in this article. Compared to cuBLAS, our better performance is because we rely on auto-tuning for the particular input size, while cuBLAS uses hand-crafted heuristics optimized toward average high-performance over various sizes, thereby avoiding the auto-tuning overhead. However, auto-tuning is an amortized one-time overhead in many application areas. For example, in siamese, the same 25 input sizes (listed in Table 3) are reused in each program run, such that auto-tuning becomes an acceptable one-time overhead per target architecture only.

8 COMPARISON TO RELATED WORK

Auto-tuning approaches can be classified into two major categories: (1) *special-purpose* systems which are designed toward a particular application class (e.g., linear algebra routines or stencil computations), and (2) *general-purpose* frameworks which target a broad range of arbitrary (possibly emerging/upcoming) application classes.

Special-purpose auto-tuning has proved to achieve impressive tuning results for important applications, e.g., FFT [16], DSP [47], chemistry computations [7], linear algebra [33, 69], self-adaptive architectures [24], geophysics [55], code mapping [35], multi-GPU systems [54], hardware synthesis [9], compiler optimization [15, 17, 32], code variant tuning [37, 61, 62], networks-on-chip [28], stencil computations [12, 36], resource virtualization [68], loop optimization [11], shared memory multiprocessors [57], load balancing [71], threading models [26, 56], machine learning [30], graph analytics [1], dynamic parallelism [60], and execution policies [8]. However, special-purpose auto-tuning severely hinders programmer's productivity, because a special-purpose auto-tuner has to be designed and implemented for each particular application class.

The alternative approach, general-purpose auto-tuning, aims at tackling the productivity issue by providing the programmer with a general framework for conveniently generating special-purpose auto-tuning systems. The currently popular general-purpose auto-tuning frameworks are OpenTuner [2], CLTune [41], KernelTuner [66], and `libtuning` [46]. These approaches generate efficient special-purpose auto-tuners for applications whose tuning parameters are either independent of each other (OpenTuner and `libtuning`) or have small ranges (CLTune and KernelTuner). However, the approaches have weaknesses regarding auto-tuning recent parallel applications for state-of-the-art architectures, because such applications rely on interdependent tuning parameters with large ranges. OpenTuner and `libtuning` often fail for such applications because, by design, they assume tuning parameters to be independent of each other. In contrast, CLTune and KernelTuner are designed toward interdependent tuning parameters, but they struggle with large ranges for such parameters, because they rely on straightforward mechanisms for generating, storing, and exploring the search spaces of such parameters. The weaknesses of the state-of-the-art general-purpose approaches are discussed in Sections 3–5 and shown experimentally in Section 7.

Classical approaches for general-purpose auto-tuning of programs with interdependent tuning parameters are ActiveHarmony [65] and Orio [19, 38]. ActiveHarmony uses for generating its constrained search space the constraint solver that we discussed in Section 7.4. However, ActiveHarmnoy suffers from similar high search space generation time as CLTune whose time is higher than the time we present for the solver in Figure 3. This is because ActiveHarmony internally relies on search space constraints, similarly as CLTune, thereby hindering the solver from achieving its full performance potential (which is still lower than ATF's efficiency for search space generation, as shown in Figure 3). Furthermore, ActiveHarmony suffers from high memory footprint and a time-intensive search space exploration process. This is because ActiveHarmony uses search techniques to explore the unconstrained search space and whenever an invalid configuration is found, it maps the configuration to a valid configuration in its (previously generated) constrained search space based on *A* pproximate Nearest Neighbor (ANN) search [3]. However, ANN has two major drawbacks when used in auto-tuning: (1) high memory footprint: for `d` tuning parameters and a search space size of `n`, ANN requires `O(d·n)` memory space, similarly as CLTune, while ATF's memory footprint is usually substantially less (Figure 4); (2) time-intensive initialization: ANN requires additional, significant initialization time, of `O(d·n·log(n))`, for preparing its internal data structures. In contrast to ActiveHarmnony, we introduce an exploration strategy for ATF toward directly exploring the constrained search space, based on our novel chain-of-trees search space structure from Section 4, thereby avoiding the memory and time-intensive process

of ANN. The other classical approach Orio supports interdependent parameters by exploring the unconstrained search space (similarly as OpenTuner and `libtuning`) and setting a penalty value for invalid configurations; thereby, Orio avoids generating and storing the entire search space [38] as inherently required by our ATF approach (as well as CLTune and KernelTuner). However, by relying on the unconstrained search space, Orio suffers from the same weaknesses as discussed and experimentally shown in this article for OpenTuner and `libtuning`.

We present the ATF which addresses the weaknesses in state-of-the-art general-purpose auto-tuning for programs with interdependent tuning parameters. ATF was originally introduced in previous work—as both an online approach based on a C++ user interface [50], as well as an offline approach that relies on a Domain-Specific Language (DSL) for auto-tuning [49]—where its user interface and convenient usage are presented. For example, the previous work shows that ATF's user interface provides (at least) the same user experience as compared to the interfaces of the state-of-the-art frameworks CLTune and OpenTuner.

In contrast to the previous work about ATF, we introduce in this article novel, optimized mechanisms for *generating*, *storing*, and *exploring* the search space of interdependent tuning parameters (constrained search space). The former implementation of ATF [49, 50] relies on only straightforward, prototype mechanisms for these three main phases of auto-tuning (not presented or published in the previous work); thereby, the former ATF is unfeasible for important applications, as confirmed experimentally in Section 7. In particular, we introduce a novel chain-of-trees search space structure which significantly reduces memory footprint of constrained search spaces, thereby enabling auto-tuning important parallel applications within the memory limitations of state-of-the-art parallel systems (Figure 4). Moreover, our novel chain-of-trees structure significantly improves exploring large search spaces (Figure 5). We also present an improved algorithm for generating constrained search spaces more efficiently than in ATF's prototype implementation [49, 50] by (i) exploiting parameter constraints on each particular tuning parameter, rather than only on a group of interdependent parameters, and (ii) parallelizing the generation of the individual search space parts of an interdependent parameter group. Our experiments in Section 7 confirm that ATF's novel mechanisms for search space generation, optimization, and exploration—the key contributions of this work—significantly improve the auto-tuning efficiency of the former ATF; thereby, we combine auto-tuning efficiency (this article) with productivity (previous work [49, 50]).

Our new mechanisms are especially important for online auto-tuning where auto-tuning is performed at program runtime: our novel search space generation mechanism contributes to significantly lower program initialization time than competitors (Figure 3), because in online auto-tuning, the search space is usually generated at program start based on runtime values (e.g., the input size). Moreover, our exploration mechanism finds well-performing configurations faster (Figure 5), thereby further contributing to lower program runtime. Related approaches like ActiveHarmony when used for the special case of online auto-tuning rely on penalty values to avoid their time-intensive processes of constrained search space generation and exploration. However, relying on penalty values is unfeasible when aiming at auto-tuning modern parallel applications, as we discussed in detail and showed experimentally in this article with the examples of OpenTuner and `libtuning`.

9 CONCLUSION

The ATF is a general-purpose auto-tuning approach for programs with interdependent tuning parameters. This article presents novel mechanisms for ATF toward efficiently generating, storing, and exploring the search spaces of such parameters. Compared to the state-of-the-art general-purpose auto-tuning frameworks, ATF's new contributions improve each particular phase of the

auto-tuning process: (1) ATF generates the search spaces of interdependent tuning parameters faster, (2) ATF requires less memory for storing these spaces, and (3) ATF achieves a higher exploration efficiency for such spaces. Our experiments confirm that ATF substantially enhances general-purpose auto-tuning as compared to the state of the art, and it enables efficiently auto-tuning applications from popular application domains, including stencil computations, linear algebra routines, quantum chemistry computations, and data mining algorithms.

REFERENCES

[1] M. Ahmad and O. Khan. 2016. GPU concurrency choices in graph analytics. In *2016 IEEE International Symposium on Workload Characterization (IISWC'16)*. 1–10.

[2] 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. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*. ACM, 303–316.

[3] Sunil Arya, David M. Mount, Nathan S. Netanyahu, Ruth Silverman, and Angela Y. Wu. 1998. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *J. ACM* 45, 6 (Nov. 1998), 891–923. DOI: https://doi.org/10.1145/293347.293348

[4] ATF Artifact Implementation. 2020. Retrieved from https://gitlab.com/mdh-project/taco2020-atf.

[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* 106, 11 (Nov. 2018), 2068–2083. DOI: https://doi.org/10.1109/JPROC.2018.2841200

[6] Protonu Basu, Mary Hall, Malik Khan, Suchit Maindola, Saurav Muralidharan, Shreyas Ramalingam, Axel Rivera, Manu Shantharam, and Anand Venkat. 2013. Towards making autotuning mainstream. *Int. J. High Performance Comput. Appl.* 27, 4 (2013), 379–393. DOI: https://doi.org/10.1177/1094342013493644

[7] Gerald Baumgartner, Alexander Auer, David E. Bernholdt, Alina Bibireata, Venkatesh Choppella, Daniel Cociorva, Xiaoyang Gao, Robert J. Harrison, So Hirata, Sriram Krishnamoorthy, et al. 2005. Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models. *Proc. IEEE* 93, 2 (2005), 276–292.

[8] David Beckingsale, Olga Pearce, Ignacio Laguna, and Todd Gamblin. 2017. Apollo: Reusable models for fast, dynamic tuning of input-dependent code. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS'17)*. IEEE, 307–316.

[9] João M. P. Cardoso, Tiago Carvalho, José G. F. Coutinho, Ricardo Nobre, Razvan Nane, Pedro C. Diniz, Zlatko Petrov, Wayne Luk, and Koen Bertels. 2013. Controlling a complete hardware synthesis toolchain with LARA aspects. *Microprocess. Microsyst.* 37, 8, Part C (2013), 1073–1089. DOI: https://doi.org/10.1016/j.micpro.2013.06.001 Special Issue on European Projects in Embedded System Design: EPESD2012.

[10] Cedric Nugteren. 2020. CLTune Issue. Retrieved from https://github.com/CNugteren/CLTune/blob/master/src/searchers/annealing.cc#L134 (commit: 2b49667).

[11] Chun Chen, Jacqueline Chame, and Mary Hall. 2008. *CHiLL: A Framework for Composing High-Level Loop Transformations*. Technical Report. Citeseer. 0–27 pages.

[12] Matthias Christen, Olaf Schenk, and Helmar Burkhart. 2011. PATUS: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *2011 IEEE International Parallel & Distributed Processing Symposium*. IEEE, 676–687.

[13] Marco Cianfriglia, Flavio Vella, Cedric Nugteren, Anton Lokhmotov, and Grigori Fursin. 2018. A model-driven approach for a new generation of adaptive libraries. *CoRR* abs/1806.07060 (2018), 14 pp. arxiv:1806.07060 http://arxiv.org/abs/1806.07060.

[14] T. Daniel Crawford and Henry F. Schaefer. 2000. An introduction to coupled cluster theory for computational chemists. *Revi. Comput. Chem.* 14 (2000), 33–136.

[15] Christophe Dubach, John Cavazos, Björn Franke, Grigori Fursin, Michael F. P. O'Boyle, and Olivier Temam. 2007. Fast compiler optimisation evaluation using code-feature based performance prediction. In *Proceedings of the 4th International Conference on Computing Frontiers*. ACM, 131–142.

[16] Matteo Frigo and Steven G. Johnson. 2005. The design and implementation of FFTW3. *Proc. IEEE* 93, 2 (2005), 216–231.

[17] Grigori Fursin, Yuriy Kashnikov, Abdul Wahid Memon, Zbigniew Chamski, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Bilha Mendelson, Ayal Zaks, Eric Courtois, et al. 2011. Milepost GCC: Machine learning enabled self-tuning compile. *Int. J. Parallel Program.* 39, 3 (2011), 296–327.

[18] Bastian Hagedorn, Larisa Stoltzfus, Michel Steuwer, Sergei Gorlatch, and Christophe Dubach. 2018. High performance stencil code generation with lift. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization (CGO'18)*. ACM, New York, NY, 100–112. DOI: https://doi.org/10.1145/3168824

[19] Albert Hartono, Boyana Norris, and Ponnuswamy Sadayappan. 2009. Annotation-based empirical performance tuning using Orio. In *2009 IEEE International Symposium on Parallel & Distributed Processing*. IEEE, 1–11.

[20] K. Hentschel et al. 2008. *Das Krebsregister-Manual der Gesellschaft der epidemiologischen Krebsregister in Deutschland e.V.* Zuckschwerdt Verlag.

[21] Intel. 2020. Math Kernel Library. Retrieved from https://software.intel.com/en-us/mkl.

[22] Intel. 2020. Math Kernel Library for Deep Learning Networks. Retrieved from https://software.intel.com/en-us/articles/intel-mkl-dnn-part-1-library-overview-and-installation.

[23] ISO/IEC. 2017. ISO international standard ISO/IEC 14882:2017—Programming language C++.

[24] B. Janßen, F. Schwiegelshohn, M. Koedam, F. Duhem, L. Masing, S. Werner, C. Huriaux, A. Courtay, E. Wheatley, K. Goossens, F. Lemonnier, P. Millet, J. Becker, O. Sentieys, and M. Hübner. 2015. Designing applications for heterogeneous many-core architectures with the FlexTiles Platform. In *2015 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS'15)*. 254–261.

[25] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM International Conference on Multimedia*. ACM, 675–678.

[26] Z. Jia, C. Xue, G. Chen, J. Zhan, L. Zhang, Y. Lin, and P. Hofstee. 2016. Auto-tuning Spark big data workloads on POWER8: Prediction-based dynamic SMT threading. In *2016 International Conference on Parallel Architecture and Compilation Techniques (PACT'16)*. 387–400.

[27] K. Kaszyk, H. Wagstaff, T. Spink, B. Franke, M. O'Boyle, B. Bodin, and H. Uhrenholt. 2019. Full-system simulation of mobile CPU/GPU platforms. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'19)*. 68–78. DOI: https://doi.org/10.1109/ISPASS.2019.00015

[28] A. E. Kiasari, Z. Lu, and A. Jantsch. 2013. An analytical latency model for networks-on-chip. *IEEE Trans. Very Large Scale Integration (VLSI) Syst.* 21, 1 (2013), 113–123.

[29] Jinsung Kim, Aravind Sukumaran-Rajam, Vineeth Thumma, Sriram Krishnamoorthy, Ajay Panyala, Louis-Noël Pouchet, Atanas Rountev, and P. Sadayappan. 2019. A code generator for high-performance tensor contractions on GPUs. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO'19)*. IEEE Press, Piscataway, NJ, 85–95. http://dl.acm.org/citation.cfm?id=3314872.3314885.

[30] Patrick Koch, Oleg Golovidov, Steven Gardner, Brett Wujek, Joshua Griffin, and Yan Xu. 2018. Autotune: A derivative-free optimization framework for hyperparameter tuning. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD'18)*. Association for Computing Machinery, New York, NY, 443–452. DOI: https://doi.org/10.1145/3219819.3219837

[31] Bastian Köpcke, Michel Steuwer, and Sergei Gorlatch. 2019. Generating efficient FFT GPU code with lift. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Functional High-Performance and Numerical Computing (FHPNC'19)*. ACM, New York, NY, 1–13. DOI: https://doi.org/10.1145/3331553.3342613

[32] Prasad Kulkarni, Stephen Hines, Jason Hiser, David Whalley, Jack Davidson, and Douglas Jones. 2004. Fast searches for effective optimization phase sequences. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI'04)*. Association for Computing Machinery, New York, NY, 171–182. DOI: https://doi.org/10.1145/996841.996863

[33] Junjie Lai and André Seznec. 2012. Bound the peak performance of SGEMM on GPU with software-controlled fast memory. *[Research Report] RR-7923, 2012. hal-00686006v1.*

[34] John Lawson, Mehdi Goli, Duncan McBain, Daniel Soutar, and Louis Sugy. 2019. Cross-platform performance portability using highly parametrized SYCL kernels. *CoRR* abs/1904.05347 (2019), 11 pp. arxiv:1904.05347 http://arxiv.org/abs/1904.05347

[35] Alberto Magni, Dominik Grewe, and Nick Johnson. 2013. Input-aware auto-tuning for directive-based GPU programming. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units (GPGPU-6)*. Association for Computing Machinery, New York, NY, 66–75. DOI: https://doi.org/10.1145/2458523.2458530

[36] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. 2015. PolyMage: Automatic optimization for image processing pipelines. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15)*. Association for Computing Machinery, New York, NY, 429–443. DOI: https://doi.org/10.1145/2694344.2694364

[37] Saurav Muralidharan, Manu Shantharam, Mary Hall, Michael Garland, and Bryan Catanzaro. 2014. Nitro: A framework for adaptive code variant tuning. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE, 501–512.

[38] T. Nelson, A. Rivera, P. Balaprakash, M. Hall, P. D. Hovland, E. Jessup, and B. Norris. 2015. Generating efficient tensor contractions for GPUs. In *2015 44th International Conference on Parallel Processing*. 969–978.

[39] Gustavo Niemeyer. 2018. Python-constraint. Retrieved from https://pypi.org/project/python-constraint/.

[40] Cedric Nugteren. 2018. CLBlast: A tuned OpenCL BLAS library. In *Proceedings of the International Workshop on OpenCL*. ACM, 1–10.

[41] Cedric Nugteren and Valeriu Codreanu. 2015. CLTune: A generic auto-tuner for OpenCL kernels. In *2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip*. IEEE, 195–202.

[42] NVIDIA. 2020. cuBLAS library. Retrieved from https://developer.nvidia.com/cublas.

[43] NVIDIA. 2020. CUDA C++ Best Practices Guide. Retrieved from https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html.

[44] NVIDIA. 2020. CUDA®Deep Neural Network library. Retrieved from https://developer.nvidia.com/cudnn.

[45] OpenTuner. 2018. Interdependent Tuning Parameters (Issue 106). Retrieved from https://github.com/jansel/opentuner/issues/106.

[46] Philip Pfaffe, Tobias Grosser, and Martin Tillmann. 2019. Efficient hierarchical online-autotuning: A case study on polyhedral accelerator mapping. In *Proceedings of the ACM International Conference on Supercomputing (ICS'19)*. ACM, New York, NY, 354–366. DOI: https://doi.org/10.1145/3330345.3330377

[47] Markus Puschel, José M. F. Moura, Jeremy R. Johnson, David Padua, Manuela M. Veloso, Bryan W. Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, et al. 2005. SPIRAL: Code generation for DSP transforms. *Proc. IEEE* 93, 2 (2005), 232–275.

[48] Ari Rasch and Sergei Gorlatch. 2018. Multi-dimensional homomorphisms and their implementation in OpenCL. *Int. J. Parallel Program.* 46, 1 (01 Feb. 2018), 101–119. DOI: https://doi.org/10.1007/s10766-017-0508-z

[49] Ari Rasch and Sergei Gorlatch. 2019. ATF: A generic, directive-based auto-tuning framework. *Concurrency Comput.: Pract. Exper.* 31, 5 (2019), 1–14.

[50] A. Rasch, M. Haidl, and S. 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. DOI: https://doi.org/10.1109/HPCC-SmartCity-DSS.2017.9

[51] A. Rasch, R. Schulze, and S. Gorlatch. 2019. Generating portable high-performance code via multi-dimensional homomorphisms. In *28th International Conference on Parallel Architectures and Compilation Techniques (PACT'19)*. 354–369.

[52] Ari Rasch, Richard Schulze, Waldemar Gorus, Jan Hiller, Sebastian Bartholomäus, and Sergei Gorlatch. 2019. High-performance probabilistic record linkage via multi-dimensional homomorphisms. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing (SAC'19)*. Association for Computing Machinery, New York, NY, 526–533. DOI: https://doi.org/10.1145/3297280.3297330

[53] Simon Rovder, José Cano, and Michael O'Boyle. 2019. Optimising convolutional neural networks inference on low-powered GPUs. In *12th International Workshop on Programmability and Architectures for Heterogeneous Multicores (MULTIPROG-2019)*. 14 pp.

[54] D. Schaa and D. Kaeli. 2009. Exploring the multiple-GPU design space. In *2009 IEEE International Symposium on Parallel Distributed Processing*. 1–12.

[55] Mohammed Sourouri, Espen Birger Raknes, Nico Reissmann, Johannes Langguth, Daniel Hackenberg, Robert Schöne, and Per Gunnar Kjeldsberg. 2017. Towards fine-grained dynamic tuning of HPC applications on modern multi-core architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 1–12.

[56] Akshitha Sriraman and Thomas F. Wenisch. 2018. μTune: Auto-tuned threading for OLDI microservices. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*. USENIX Association, Carlsbad, CA, 177–194. https://www.usenix.org/conference/osdi18/presentation/sriraman.

[57] Per Stenström and Jonas Skeppstedt. 1997. A performance tuning approach for shared-memory multiprocessors. In *Euro-Par'97 Parallel Processing*, Christian Lengauer, Martin Griebl, and Sergei Gorlatch (Eds.). Springer, Berlin, 72–83.

[58] Larisa Stoltzfus, Bastian Hagedorn, Michel Steuwer, Sergei Gorlatch, and Christophe Dubach. 2019. Tiling optimizations for stencil computations using rewrite rules in lift. *ACM Trans. Archit. Code Optim.* 16, 4, (Dec. 2019), Article 52, 25 pages. DOI: https://doi.org/10.1145/3368858

[59] Huihui Sun, Florian Fey, Jie Zhao, and Sergei Gorlatch. 2019. WCCV: Improving the vectorization of IF-statements with warp-coherent conditions. In *Proceedings of the ACM International Conference on Supercomputing (ICS'19)*. ACM, New York, NY, 319–329. DOI: https://doi.org/10.1145/3330345.3331059

[60] X. Tang, A. Pattnaik, H. Jiang, O. Kayiran, A. Jog, S. Pai, M. Ibrahim, M. T. Kandemir, and C. R. Das. 2017. Controlled kernel launch for dynamic parallelism in GPUs. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA'17)*. 649–660. DOI: https://doi.org/10.1109/HPCA.2017.14

[61] Thiago SFX Teixeira, William Gropp, and David Padua. 2019. Managing code transformations for better performance portability. *Int. J. High Performance Comput. Appl.* 33, 6 (2019), 1290–1306.

[62] Thiago S. F. X. Teixeira, Corinne Ancourt, David Padua, and William Gropp. 2019. Locus: A system and a language for program optimization. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO'19)*. IEEE Press, Piscataway, NJ, 217–228.

[63] Philippe Tillet and David Cox. 2017. Input-aware auto-tuning of compute-bound HPC kernels. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 1–12.

[64] Philippe Tillet, H. T. Kung, and David Cox. 2019. Triton: An intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL'19)*. ACM, New York, NY, 10–19. DOI: https://doi.org/10.1145/3315508.3329973

[65] Ananta Tiwari, Vahid Tabatabaee, and Jeffrey K. Hollingsworth. 2009. Tuning parallel applications in parallel. *Parallel Comput.* 35, 8 (2009), 475–492. DOI: https://doi.org/10.1016/j.parco.2009.07.001

[66] Ben van Werkhoven. 2019. Kernel tuner: A search-optimizing GPU code auto-tuner. *Future Gen. Comput. Syst.* 90 (2019), 347–358. DOI: https://doi.org/10.1016/j.future.2018.08.004

[67] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary Devito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2019. The next 700 accelerated layers: From mathematical expressions of network computation graphs to accelerated GPU kernels, automatically. *ACM Trans. Archit. Code Optim.* 16, 4 (Oct. 2019), Article 38, 26 pages. DOI: https://doi.org/10.1145/3355606

[68] N. Vijaykumar, K. Hsieh, G. Pekhimenko, S. Khan, A. Shrestha, S. Ghose, A. Jog, P. B. Gibbons, and O. Mutlu. 2016. Zorua: A holistic approach to resource virtualization in GPUs. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*. 1–14.

[69] R. Clinton Whaley and Jack J. Dongarra. 1998. Automatically tuned linear algebra software. In *SC'98: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*. IEEE, 38.

[70] Stephen Wright and Jorge Nocedal. 1999. Numerical optimization. *Springer Sci.* 35, 67–68 (1999), 7.

[71] Vasileios Zois, Divya Gupta, Vassilis J. Tsotras, Walid A. Najjar, and Jean-Francois Roy. 2018. Massively parallel skyline computation for processing-in-memory architectures. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques (PACT'18)*. Association for Computing Machinery, New York, NY, Article 1, 12 pages. DOI: https://doi.org/10.1145/3243176.3243187

Received May 2020; revised September 2020; accepted September 2020