pyATF: Constraint-Based Auto-Tuning in Python

Richard Schulze
University of Münster
Muenster, Germany
r.schulze@uni-muenster.de

Sergei Gorlatch
University of Münster
Muenster, Germany
gorltach@uni-muenster.de

Ari Rasch
University of Münster
Muenster, Germany
a.rasch@uni-muenster.de

Abstract

We introduce pyATF – a new, language-independent, open-source auto-tuning tool that fully automatically determines optimized values of performance-critical program parameters. A major feature of pyATF is its support for constrained parameters, e.g., the value of one parameter has to divide the value of another parameter. A further major feature of pyATF is its user interface which is designed with a particular focus on expressivity and usability for real-world demands, and which is offered in the increasingly popular Python programming language. We experimentally confirm the practicality of pyATF using real-world studies from the areas of quantum chemistry, image processing, data mining, and deep learning: we show that pyATF auto-tunes the complex parallel implementations of our studies to higher performance than achieved by state-of-practice approaches, including hand-optimized vendor libraries.

CCS Concepts: • **Software and its engineering → Compilers**.

Keywords: auto-tuning, constraints, CUDA, OpenCL

ACM Reference Format:
Richard Schulze, Sergei Gorlatch, and Ari Rasch. 2025. pyATF: Constraint-Based Auto-Tuning in Python. In *Proceedings of the 34th ACM SIGPLAN International Conference on Compiler Construction (CC '25), March 1–2, 2025, Las Vegas, NV, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3708493.3712682

1 Introduction

Auto-tuning [9] is a popular technique for automatically optimizing programs for a particular target architecture and characteristics of the input and output data (e.g., size and memory layout) [73]. To use auto-tuning, the user implements – or a compiler generates – the target program as generic in performance-critical parameters (a.k.a. *tuning parameters*), such as sizes of tiles and numbers of threads, and an auto-tuning tool then fully automatically identifies an architecture- and data-optimized configuration of these parameters.

CC '25, Las Vegas, NV, USA

ACM ISBN 979-8-4007-1407-8/25/03
https://doi.org/10.1145/3708493.3712682

Framework	Year	API	Constr.	Targets
OpenTuner [6]	2014	Python	×	∗
CLTune [47]	2015	C++	(✓)	OpenCL
Kernel Tuner [75]	2019	Python	(✓)	OpenCL,CUDA, . . .
HyperMapper [45]	2019	JSON	×	∗
KTT [50]	2020	C++	(✓)	OpenCL,CUDA, . . .
ytopt [79]	2021	Python	(✓)	∗
ATF [61]	2021	DSL	✓	∗
BaCO [29]	2023	JSON	✓	∗
pyATF (this work)	**2024**	**Python**	**✓**	**∗**

Figure 1. State-of-the-art generic auto-tuning tools (including pyATF). The symbol × means that constraints are not supported, and symbol "∗" denotes that programs in any target programming language can be auto-tuned.

Auto-tuning has already been successfully used in various important application domains, including linear algebra [77] and stencil computations [17], and it can be broadly classified into two categories: 1) the traditional, *non-generic* tools which are specifically designed and tied to a particular application domain (e.g., only linear algebra [77]), and 2) the recent, *generic* auto-tuning tools that can be used for arbitrary kinds of domains.

Our work is inspired by generic auto-tuning approaches due to their broad applicability, particularly in the area of compiler construction [9]. An overview of the state-of-the-art generic auto-tuning tools is given in Figure 1.

Each generic auto-tuner offers the user an interface in a particular *API (Application Programming Interface)*, e.g., a Python-based API (as OpenTuner) or an API based on C++ (as CLTune). Using the API, the user defines the search space of the program to be tuned, starts the auto-tuning process, sets an abort condition that stops the tuning process (e.g., a time interval), etc.

Generic auto-tuners also differ in terms of their support for so-called *constraints* on tuning parameters [61]. Efficiently supporting constraints has proven to be essential for auto-tuning contemporary parallel implementations that target state-of-the-art parallel architectures [29, 61], e.g., due to the hierarchically arranged layers of memories and cores of these architectures. For example, the value of a tile size tuning parameter on an upper memory layer usually has to be constrained as a multiple of a tile size for a lower memory layer, because a lower-layer tile is a chunk of an upper-layer tile. Rasch et al. [61] introduce optimized concepts for implementing efficient, constraint-supporting auto-tuning tools; these

concepts have been implemented in the ATF tuner [61] (denoted via a check mark symbol in Figure 1) and been adopted by BaCO [29] to achieve high auto-tuning efficiency for constrained tuning parameters. In contrast, earlier approaches, e.g., CLTune, have only limited support for constraints (they support only small search spaces when constraints are used, as we discuss later – denoted by parentheses around check mark symbols in Figure 1), and approaches OpenTuner and HyperMapper have no support for constraints at all, which severely limits their applicability (as we also discuss later).

Auto-tuners and their interfaces also differ in terms of their generality. For example, while OpenTuner can be used for auto-tuning programs implemented in arbitrary programming languages (indicated via symbol ⋆ in Figure 1), CLTune is specifically designed for auto-tuning OpenCL programs only.

We introduce pyATF[1,2] – a new, generic, publicly available open-source auto-tuning tool whose *interface is designed with a particular focus on usability and expressivity for real-world demands* (first major contribution), and which *combines three major advantages of the state-of-the-art auto-tuning tools listed in Figure 1* (second major contribution):

1. pyATF's user interface is designed in the Python programming language which is becoming increasingly important in both academia and industry, e.g., due to its ease of use even for non-expert users [1];
2. pyATF efficiently supports constraints on tuning parameters – by internally implementing and relying on state-of-the-art auto-tuning techniques [61] – thereby enabling an efficient auto-tuning process for modern parallel implementations;
3. pyATF is designed as generic in the target programming language such that it can be used for auto-tuning programs written in arbitrary programming languages.

Our experiments on GPU and CPU confirm that pyATF achieves high auto-tuning efficiency compared to the state-of-the-art auto-tuning tools listed in Figure 1, using real-world case studies from important domains: quantum chemistry, image processing, data mining, and deep learning.

The paper is structured as follows. After summarizing the state-of-the-art generic auto-tuning tools in Section 2, we illustrate pyATF's design and functionality in Section 3, and we argue that pyATF is easier to use and more expressive than the current state-of-the-art generic auto-tuning tools. We present experimental results in Section 4, discuss related work in Section 5, and we conclude in Section 6.

2 State of the Art in Generic Auto-Tuning

Our pyATF tool aims to i) combine the advantages of the state-of-the-art auto-tuning tools listed in Figure 1 and ii) hide these advantages behind a new, productive, and user-friendly interface for auto-tuning (which is described in Section 3).

[1]pyATF is documented and available open source: https://atf-tuner.org
[2]We call our tool pyATF, because i) it is based on python, and ii) it internally uses the same theoretical auto-tuning foundation as ATF [61].

```
__kernel void saxpy( const          int     N,   1
                     const          float   a,   2
                     const __global float*  x,   3
                           __global float*  y    4
                   )                            5
{                                               6
  for( int w = 0; w < WPT; ++w )                7
  {                                             8
    const int index = w * get_global_size(0)    9
                        + get_global_id(0);     10
                                                11
    y[ index ] += a * x[ index ];               12
  }                                             13
}                                               14
```

Listing 1. Auto-tunable SAXPY kernel from the OpenCL CLBlast library [46] (simplified)

In the following, we highlight the advantages of the state-of-the-art generic auto-tuning tools – in terms of their: 1) *API*, 2) support for *Constraints*, and 3) supported *Targets* – see Figure 1. For this, we focus on BaCO and ytopt, because BaCO is the latest auto-tuning tool that *supports constraints* on tuning parameters, and ytopt is the latest auto-tuner whose *API is based on Python* – both tools support programs written in *arbitrary target programming languages*.

We illustrate the user interfaces of BaCO and ytopt (to compare them later with pyATF's interface) using the simple, illustrative example SAXPY of the auto-tunable *CLBlast* OpenCL library [46]. The SAXPY computation takes as arguments the input size N, a floating-point value a, and two N-sized vectors x and y of floating-point values, and it computes for all $i \in \{1, \ldots, N\}$:

$$y[\,i\,] = a * x[\,i\,] + y[\,i\,]$$

Listing 1 shows CLBlast's OpenCL implementation of SAXPY. The kernel is executed on a device (e.g., a GPU) in parallel by several *Work-Items (WIs)* — the OpenCL term for thread. Each WI computes a WPT-sized tile (line 7) of the result vector – WPT is a tuning parameter and thus has to be replaced by a concrete value that is optimized for the particular target architecture and input size N (e.g., using one of the auto-tuning tools listed in Figure 1). The WIs iterate over their corresponding tile of the input (line 7), and each WI computes in each iteration the index of the input elements of x and y (line 9) to be used in the computation of SAXPY (line 12). OpenCL requires work-items to be grouped into *Work-Groups (WGs)* [34]. The number of work-items per work-group is called *Local Size (*LS*)* which is a further tuning parameter. The local size is set in the so-called *host code* [34] (not presented for brevity) which is used in OpenCL to invoke the kernel on a device.

For the correctness of the SAXPY kernel, WPT must divide the input size N – a constraint on the WPT tuning parameter – such that each WI processes an equally sized tile of the input. Moreover, the OpenCL specification requires the local size LS to divide the *Global Size (*GS*)* – a constraint on tuning parameter LS. The GS is the total number of work-items in OpenCL, and it is set as N/WPT in the case of the SAXPY kernel. Analogously to the local size, the global size is set in the host code when invoking the kernel.

2.1 Constraint-Supporting Auto-Tuner

BaCO [29] is the latest auto-tuning tool for programs with constrained tuning parameters. It is implemented within the HyperMapper [45] framework, and it handles constraints according to the concepts by Rasch et al. [61]. In BaCO, the tuning process is specified via a JSON file, and the process is started via an additional Python program.

Listing 2 shows the BaCO JSON file for auto-tuning the SAXPY example in Listing 1; the BaCO Python program for running the JSON file is presented in Listing 3.

In Listing 2, lines 1-4 specify metainformation for BaCO, and lines 5 and 6 choose the search technique and abort condition. Afterwards, lines 7-18 define the search space based on tuning parameters: i) parameter WPT (line 8) is defined as an integer value (line 9) in the interval from 1 to the input size 1000 (line 10)[3], and it has to evenly divide the input size (line 11); ii) parameter LS (line 13) is defined similarly to parameter WPT, but has to evenly divide the input size divided by WPT (lines 16-17).

```
1  { "application_name": "saxpy",
2   "optimization_objectives": ["runtime"],
3   "output_data_file": "./saxpy.csv",
4   "log_file": "./saxpy.log",
5   "optimization_method":"bayesian_optimization",
6   "optimization_iterations": 50,
7   "input_parameters": {
8     "WPT": {
9       "parameter_type": "integer",
10      "values": [1, 1000],
11      "constraints": ["1000 % WPT == 0"],
12      "dependencies": [] },
13    "LS": {
14      "parameter_type": "integer",
15      "values": [1, 1000],
16      "constraints": ["(1000 / WPT) % LS == 0"],
17      "dependencies": ["WPT"]
18    } } }
```

Listing 2. BaCO JSON tuning file for auto-tuning SAXPY

Listing 3 shows the corresponding BaCO Python program which implements a *cost function* (lines 2-4) that compiles, runs, and measures the runtime of the SAXPY kernel for a particular tuning configuration. The program starts the tuning process (line 7) based on the BaCO JSON file in Listing 2.

[3]BaCO requires hard coding the input size because it relies on JSON.

```
# Implement a Cost Function                          1
def cost_function(config):                           2
    # ... SAXPY Kernel & Host Code (>50 LOC)         3
    return {'valid': valid, 'runtime': runtime}      4
                                                     5
# Generate & Explore the Search Space                6
hypermapper.optimize('spec.json', cost_function)     7
```

Listing 3. BaCO Python program for running the tuning file in Listing 2

2.2 Python-Based Auto-Tuner

The ytopt [79] tool is the latest auto-tuner that offers a Python-based user interface.

Listing 4 shows the ytopt Python program for auto-tuning the OpenCL SAXPY example in Listing 1. The search space is defined using tuning parameters (lines 5-14 of Listing 4), analogously as in the BaCO program in Listing 2. However, in contrast to BaCO, the ytopt tool defines constraints (lines 13-14 in Listing 4) on entire parameter configurations, whereas BaCO defines constraints on individual parameters (Listing 2, lines 11 and 16) – this ATF-inspired constraint design [61] allows BaCO to internally use the optimized processes of search space generation, storing, and exploration introduced by Rasch et al. [61]. Afterwards, in lines 17-19 of Listing 4, the cost function for SAXPY is defined (the same as in the BaCO program in Listing 3, lines 2-4), and ytopt's exploration process is specified in lines 22-26 of Listing 4.

```
# Input Size                                                  1
N = 1000                                                      2
                                                              3
# Generate the Search Space                                   4
cs = CCS.ConfigurationSpace()                                 5
WPT = CCS.IntNumericalParameter(name='WPT',                   6
                                lower=1   ,                   7
                                upper=N+1 )                   8
LS  = CCS.IntNumericalParameter(name='LS',                    9
                                lower=1   ,                   10
                                upper=N+1 )                   11
cs.add_parameters([WPT, LS])                                  12
cs.add_forbidden_clause(f'({N}%WPT) != 0'))                   13
cs.add_forbidden_clause(f'(({N}/WPT)%LS) != 0')               14
                                                              15
# Implement a Cost Function                                   16
def cost_function(config):                                    17
    # ... SAXPY Kernel & Host Code (>50 LOC)                  18
    return runtime                                            19
                                                              20
# Specifiy the Tuning Problem                                 21
Problem = TuningProblem(                                      22
    task_space=None,                                          23
    input_space=cs,                                           24
    output_space=Space([Real(0.0,inf)]),                      25
    objective=cost_function                                   26
)                                                             27
```

Listing 4. ytopt Python program for auto-tuning SAXPY

3 pyATF: Design and Illustration[4]

Listing 5 demonstrates how pyATF is used for auto-tuning the SAXPY kernel in Listing 1. The pyATF program in Listing 5 is written in Python and performs three main steps required for auto-tuning.

In the following, we explain the pyATF program in Listing 5, step by step, and we argue that pyATF is easier to use and more expressive (e.g., regarding expressing constraints and setting abort conditions) than the two existing, state-of-the-art auto-tuning tools BaCO (Listings 2 and 3) and ytopt (Listing 4).

3.1 Step 1: Generating the Search Space

Analogously to BaCO and ytopt (Listings 2 and 4), pyATF generates the search space based on user-defined tuning parameters (Listing 5, lines 5-10). To efficiently handle and manage the search space of constrained tuning parameters, pyATF is designed to define constraints on parameters [61] (lines 7 and 10), whereas ytopt defines constraints on entire configurations (see Section 2.2). While this design decision of ytopt does not complicate expressing constraints for the user, it severely limits ytopt's auto-tuning efficiency [61]. BaCO uses the same constraint design as pyATF, but BaCO is limited to constraint functions implemented in *NumExpr* [24], whereas pyATF is more general and flexible by allowing any arbitrary Python callable as constraint function.

Regarding parameter ranges, we have designed pyATF to support – additionally to the interval range type (lines 6 and 9 in Listing 5) – the Set type which may contain also non-numerical values: for example, `Set('red','blue')` represents the parameter range containing the two strings red, and blue. For numerical values, the set range type is beneficial when parameter's range is small and/or does not follow a regular pattern, e.g., `Set(3,5,7)` represents the values 3,5,7.

To further contribute to the user's productivity, pyATF's interval range type is designed to (optionally) allow *generator functions*: `Interval(START,END,STEP_SIZE,GENERATOR)`. A generator function GENERATOR can be any arbitrary Python callable that takes as input a value between START and END and that yields a value of an arbitrary type. Using a generator function has the effect that the parameter range contains the values `GENERATOR(i)` for `i` $\in$ {START, ..., END}. For example, using `Interval(1,10,generator=pow2)`, where `pow2=lambda i: 2**i`, results in the first ten powers of two. In contrast to pyATF, the BaCO framework is limited to the single generator function `log` which computes *logarithm*, and ytopt has no support for generator functions at all.

[4] Our pyATF auto-tuning tool is available open source (https://github.com/atf-tuner/pyATF) and contains all the examples discussed in this section.

```
# Input Size                                                    1
N = 1000                                                        2
                                                                3
# Step 1: Generate the Search Space                             4
WPT = TP( 'WPT'                        ,                        5
          Interval( 1,N )              ,                        6
          lambda WPT: N % WPT == 0 )                            7
LS  = TP( 'LS'                                      ,           8
          Interval( 1,N )                           ,           9
          lambda WPT,LS: (N/WPT) % LS == 0 )                    10
                                                                11
# Step 2: Implement a Cost Function                             12
saxpy_code = # ... (Listing 1)                                  13
                                                                14
N = np.int32( N )                                               15
a = np.float32( np.random.random() )                            16
x = np.random.rand(N).astype(np.float32)                        17
y = np.random.rand(N).astype(np.float32)                        18
                                                                19
cf = opencl.CostFunction( saxpy_code )                          20
           .platform_id( 0 )                                 \  21
           .device_id( 0 )                                   \  22
           .kernel_args( N, a,x,y )                          \  23
           .glb_size( lambda WPT,LS: N/WPT ) \                  24
           .lcl_size( lambda     LS: LS    )                    25
                                                                26
# Step 3: Explore the Search Space                              27
config = Tuner().tuning_params( WPT,LS )    \                   28
                .search_technique( AUC() )  \                   29
                .tune( cf, Evaluations(50) )                    30
```

Listing 5. pyATF Python program for auto-tuning SAXPY

3.2 Step 2: Implementing a Cost Function

For high flexibility, pyATF supports as cost function any arbitrary Python callable, the same as BaCO and ytopt. However, implementing cost functions from scratch can be cumbersome for the user, particularly when targeting programs implemented in recent programming approaches, e.g., OpenCL and CUDA which both require the so-called *host code* [54, 62] for their execution. To simplify for the user the cumbersome task of implementing cost functions, pyATF offers pre-implemented cost functions.

Our pre-implemented `opencl.CostFunction` (Listing 5, lines 20-25) is used for conveniently auto-tuning OpenCL kernels in terms of runtime performance. In the example of Listing 5, our cost function is initialized with the OpenCL platform and device ids (lines 21-22) — this choice is arbitrary and could also be any other of system's OpenCL-compatible devices. As the kernel's arguments (line 23), we use the input size N (line 15), a random floating-point number for a (line 16), and two N-sized buffers for x and y that are also filled with random floating-point numbers (lines 17 and 18) — random data is commonly used as input when auto-tuning OpenCL kernels. The kernel's OpenCL global and local size are set in lines 24 and 25 – for high flexibility, the sizes can

also be set as arbitrary Python callables (as in the listing) to make them dependent on tuning parameter values. The initialized cost function `cf` (line 20) then takes configurations comprising concrete values of parameters `WPT` and `LS`, and it returns the `SAXPY` kernel's runtime for these concrete `WPT` and `LS` values. For this, `cf` internally replaces in the kernel's source code the tuning parameters' names (e.g., in line 7 of Listing 1) by their corresponding values in the input configuration, using the OpenCL pre-processor, and it uses pre-implemented OpenCL host code that invokes the kernel with the passed global/local size, measures and returns the kernel's runtime using the OpenCL profiling API, etc.

Our pyATF tool also provides a pre-implemented cost function for auto-tuning CUDA kernels. The function is based on the NVIDIA NVRTC [48] library and used analogously to pyATF's OpenCL cost function.

Additionally, pyATF provides a generic cost function to simplify auto-tuning programs written in an arbitrary programming language for the user, using an arbitrary tuning objective (e.g., high runtime performance and/or low energy consumption). This generic cost function is initialized with: 1) a command (that is executable on the system's command line) for compiling the program to tune, 2) a command for running the program, and, optionally, 3) the path to a cost file to which the user program writes the configuration's cost that pyATF should minimize (e.g., runtime and/or energy consumption). If no cost file is stated, pyATF automatically measures and uses program's runtime as cost. For multi-objective tuning via pyATF's generic cost function, the auto-tuned program writes comma-separated costs to the cost file; pyATF then minimizes these costs using the lexicographical order, or, alternatively, a user-defined order.

3.3 Step 3: Exploring the Search Space

The same as in `BaCO` and `ytopt`, the search space is explored in pyATF (Listing 5, lines 28-30) using a search technique, e.g., `AUC` as in Listing 5 (line 29). The tuning process is stopped when an abort condition is met, which is after 50 evaluated parameter configurations in the example of Listing 5 (line 30). We outline pyATF's supported search techniques and abort conditions in the following.

Search Techniques. Currently, pyATF offers the following, pre-implemented search techniques [78]: 1) *Differential Evolution*, 2) *Pattern Search*, 3) *Simulated Annealing*, 4) *Torczon*, 5) *Exhaustive*, and 6) *Random*. We have designed pyATF generically, i.e., new search techniques can be easily added.

Since it is often not obvious for the user which search technique to use, pyATF also pre-implements the *meta-technique* `AUC` [6] (line 29 in Listing 5). This meta-technique automatically identifies and uses a search technique for the user, by gradually testing the available search techniques and allocating more exploration time to well-performing ones – `AUC` is used as default technique in pyATF.

In contrast to pyATF, the `BaCO` and `ytopt` approaches are specifically designed and optimized for the *Bayesian* search, which often achieves good search results, but requires a careful adjustment of so-called *hyperparameters* [45], which can be complex, particularly for non-expert users. In contrast, the search techniques implemented and offered in pyATF are intended to work out-of-the-box, without requiring hyperparameter adjustments, so that auto-tuning becomes appealing also to common application developers. Furthermore, pyATF's search techniques work well for high-dimensional search spaces, whereas the Bayesian search currently has difficulties with such spaces [31], as we discuss in Section 4.

Abort Conditions. The state-of-the-art auto-tuners `BaCO` and `ytopt` support as abort condition stopping the tuning process after a user-defined number of evaluated configurations (as in line 6 of Listing 2). Additionally, `BaCO` also allows stopping the tuning process after a user-defined number of minutes: e.g., using `"time_budget": 60` to stop `BaCO`'s auto-tuning process after `60` minutes.

We have experienced, that for real-world applications, user requirements might be significantly more complex than stopping the tuning process based on tuning time only (measured in number of evaluated configurations or absolute time).

Our pyATF tuner aims to be more flexible than `BaCO` and `ytopt` regarding abort conditions, to meet complex, real-world user requirements: instead of stopping the tuning based on the tuning time only, pyATF also allows stopping the tuning process based on the tuning result (e.g., when the cost falls below a user-defined threshold) or based on both time and cost (e.g., when the cost could not be improved within a user-defined time interval).

pyATF currently supports the following pre-implemented abort conditions: 1) `Duration( timedelta(t) )` which stops the tuning process after a user-defined time interval `t`; here, `timedelta` is part of Python's `datetime` module [52], i.e., `t` can be `hours=2` or `hours=2,minutes=30`, etc; 2) `Evaluations(n)` which stops after `n` tested configurations; 3) `Fraction(f)` which stops after `f*S` tested configurations, where `f` is a floating-point value in $[0, 1]$ and `S` the search space size; 4) `Cost(c)` which stops tuning when a configuration with a cost $\leq$ `c` has been found; 5) `Speedup( s,timedelta(t) )` which stops when within the last time interval `t` the cost could not be lowered by a factor $\geq$`s`; 6) `Speedup(s,n)` which stops when within the last `n` tested configurations the cost could not be lowered by a factor $\geq$`s`. If no abort condition is set, pyATF uses `Evaluations(S)`, where `S` is the search space size.

For more complex user requirements, pyATF allows combining abort conditions via the logical operators `And`, `Or`, `Not`, e.g., `Or(Speedup(1.5,hours=3),Duration(hours=5))`. Our logical operators can be arbitrarily combined and nested to express complex abort conditions.

We have designed pyATF such that new abort conditions can be easily added, by implementing a straightforward Python interface.

4 Experimental Results[5]

This section shows that even though pyATF offers an easy-to-use, Python-based user interface (introduced in Section 3), it achieves high-quality auto-tuning results for real-world case studies.

Application Case Studies. We use four real-world case studies that differ significantly in their search space characteristics (Table 1): 1) *Coupled Cluster (*CCSD(T)*)* [18] which computes so-called molecular properties and is important in quantum chemistry, 2) *Gaussian Convolution (*CONV*)* which is a popular stencil computation and heavily used in the domain of image processing, 3) *Probabilistic Record Linkage (*PRL*)* [59] which identifies duplicate entries in a database and is from the domain of data mining, and 4) *Matrix Multiplication (*GEMM*)* which is a linear algebra routine and heavily used in deep learning.

We rely on the auto-tunable implementations of these studies that are generated according to the approach of *Multi-Dimensional Homomorphisms (MDH)* [53]. This is because the MDH-based implementations of our studies have the potential to be auto-tuned to higher performance than hand-optimized solutions [53, 58, 61], based on large and complex search spaces (described in the paragraph after next).

Data Characteristics. We use input data sizes (listed in Table 1) taken from popular real-world examples: 1) TCCG [65] benchmark suite for CCSD(T); 2) ImageNet [20] data set for CONV; 3) EKR [30], the largest cancer registry in Europe, for PRL; 4) ResNet-50 [28] neural network for GEMM.

Search Space Characteristics. The search space characteristics of our four studies are summarized in Table 1. For example, our quantum chemistry study CCSD(T) has 39 tuning parameters which have a minimum range size of 2 (boolean parameters) and a maximum range size of 24 (tile size), and the ranges contain on average 15.46 values. The search space of CCSD(T) has a size of $8.81 * 10^{18}$ – this is a small fraction of $4.41 * 10^{-25}$ of the unconstrained search space (i.e., when ignoring constraints and keeping invalid configurations within the space, as OpenTuner) which contains $> 10^{43}$ configurations. Note that the search spaces of our studies depend on the input size [53] (e.g., tile size parameters are defined in the range from 1 to the input size) to achieve high performance [73].

Table 1 confirms that our case studies have very different auto-tuning characteristics, which enables a thorough evaluation of our pyATF tool.

[5]All experiments described in this section can be reproduced using our artifact implementation [8].

Experimental Setup. Our system is equipped with two Intel Xeon Gold 6140 CPUs, 192 GB main memory, and an NVIDIA Ampere A100-PCIE-40GB GPU. We use the following versions of vendor libraries: NVIDIA cuBLAS 11.10.1 and NVIDIA cuDNN 8.9.7, as well as Intel oneMKL 2024.0.0 and Intel oneDNN 3.3.0. For TVM, we use version 0.17.0.

4.1 Comparison of Auto-Tuning Efficiency

Figure 2 reports the auto-tuning efficiency (i.e., how the runtime of the auto-tuned case studies decreases with increasing tuning time) of pyATF as compared to the state-of-the-art auto-tuning tools listed in Figure 1. For a fair comparison, we use for all tools their recommended, default search technique, e.g., AUC for pyATF, OpenTuner, and ATF. For completeness, the figure also reports the performance achieved by hand-optimized vendor libraries: NVIDIA cuDNN and Intel oneDNN for computing image processing example CONV, and NVIDIA cuBLAS and Intel oneMKL for deep-learning computation GEMM. Additionally, we also report the performance achieved by TVM [16] which relies internally on its own auto-tunable implementations[6]; the TVM-generated implementations are optimized using TVM's own Ansor [80] optimization engine which is specifically designed and optimized for TVM.

Figure 3 reports additionally for each combination of a case study and auto-tuning tool the: 1) initialization time (which is dominated by the time for generating and storing the search space [61], but also includes all runtimes for instantiating Python/C++ classes, etc); 2) search space size; 3) number of evaluated configurations that are valid (i.e., that satisfy the constraints on tuning parameters); 4) number of evaluated configurations that are invalid (that do not satisfy constraints). These characteristics often differ significantly between studies and tools, as explained in the following.

In Figure 3, the search space sizes of different tools differ for the same study. The tools pyATF and ATF generate and explore exactly the same search space, of all valid parameter configurations, based on the concepts introduced by Rasch et al. [61]. In contrast, CLTune uses a straightforward process to generate its search space [61] and thus has to rely on hand-pruned parameter ranges to generate its search space in acceptable time. However, such pruning limits CLTunes's auto-tuning efficiency (as reported in Figure 2 and discussed below)[7]. OpenTuner uses straightforwardly the space that contains also invalid configurations (which severely limits its auto-tuning efficiency, as also discussed below). KTT recently adopted the search space generation and storing techniques by Rasch et al. [61] that are also used by pyATF [2]. However, KTT has to use a smaller search space than pyATF,

[6]We rely in our experiments on MDH-based implementations [53], because these have the potential to be auto-tuned to higher performance than TVM.
[7]For a fair comparison, we use for CLTune exactly the hand-pruned parameter ranges that are recommended by the CLTune developers [47] (e.g., powers of two for tile sizes).

Table 1. Search space characteristics of our four application studies. Columns denote: 1) number of tuning parameters (#TP); 2) minimum parameter range size (Min. RS); 3) maximum parameter range size (Max. RS); 4) average parameter range size (Avg. RS); 5) search space size (|SP|); 6) fraction of the unconstrained search space that represent valid configurations (FT).

	Name	Domain	Input Size	#TP	Min. RS	Max. RS	Avg. RS	|SP|	FT
1	CCSD(T)	Quantum Chemistry	$24 \times 16 \times 16 \times 24 \times 16 \times 16 \times 24$	39	2	24	15.46	$8.81 * 10^{18}$	$4.41 * 10^{-25}$
2	CONV	Image Processing	4288×2848	15	2	4288	2379.33	$2.03 * 10^{8}$	$2.33 * 10^{-29}$
3	PRL	Data Mining	1024×1024	14	2	1024	586.14	$2.31 * 10^{7}$	$1.33 * 10^{-19}$
4	GEMM	Deep Learning	$1 \times 1000 \times 2048$	19	1	2048	642.89	$1.08 * 10^{8}$	$6.34 * 10^{-21}$

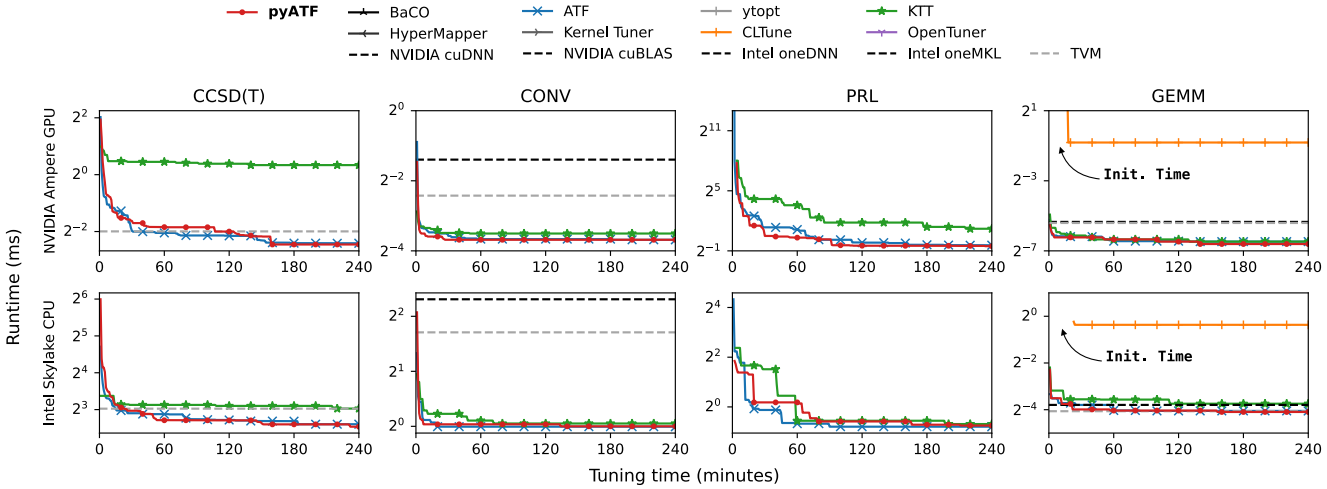

Figure 2. Auto-tuning efficiency (lower is better) of pyATF vs state-of-the-art auto-tuners (as well as hand-optimized vendor libraries and TVM) on NVIDIA GPU (top part of the figure) and Intel CPU (bottom part) for real-world case studies from popular domains. We report the medians of 10 auto-tuning runs, each run of 4h. Missing lines indicate missing support for studies: BaCO, ytopt, and HyperMapper fail for all studies, because they rely on the Bayesian search, and CLTune fails for CONV and CCSD(T), because its search space pruning causes an empty search space (CONV) or its initialization time exceeds the total auto-tuning time of 4h (CCSD(T)) — see Figure 3. TVM cannot be used for expressing PRL [7].

because KTT cannot handle tuning parameters representing OpenCL's numbers of work-groups when synchronization among work-groups is required: such synchronization is achieved in OpenCL by starting a second kernel, but KTT is inherently designed and optimized toward auto-tuning a single OpenCL kernel only. In contrast, pyATF is designed more generically (as discussed in Section 3).

Regarding initialization time in Figure 3, the tools pyATF, ATF, and KTT often have a substantially lower initialization time than CLTune, even when using the expert-recommend, hand-pruned parameter ranges for CLTune. This is because pyATF, ATF, and KTT internally implement optimized algorithms for search space generation and storing [61], which makes them applicable to important real-world studies (e.g., those in Figure 2). ATF has a lower initialization time than pyATF, because ATF is implemented in the performance-oriented C++ programming language, whereas pyATF is implemented in Python to offer a convenient user interface and to be easily extensible, e.g., by new search techniques (as discussed in Section 3)[8]. However, the differences between pyATF and ATF, e.g., ~ 40 seconds for example CONV, are negligible in general, because auto-tuning usually runs for several minutes or hours (e.g., 4h in Figure 2). OpenTuner does not explicitly generate and store a search space [61] and thus has a constant, low initialization time; however, at the cost of having invalid configurations within its space [61] which severely limits its auto-tuning efficiency, as discussed below. Initialization times differ for the same tool over different studies, because the studies have differently sized, input-dependent search spaces (as discussed above). As compared to CLTune, the KTT tool achieves a similar low initialization time as pyATF and ATF, because it also adopted the search space generation and storing techniques of Rasch et al. [61].

The number of configurations explored by pyATF and ATF differ because search techniques have a non-deterministic behavior. KTT often explores more configurations than other

[8]Note that pyATF has a lower initialization time than ATF for example CCSD(T), because our Python implementation of pyATF seems to be more efficient than ATF's C++ implementation when parameter ranges are small.

		NVIDIA Ampere GPU				Intel Skylake CPU			
		SP Size	Init. Time	Valid Configs.	Invalid Configs.	SP Size	Init. Time	Valid Configs.	Invalid Configs.
CCSD(T)	**pyATF**	8.81E+18	337 ms	5,265	0	8.81E+18	330 ms	5,507	0
	BaCO	crashes due to Bayesian search				crashes due to Bayesian search			
	ATF	8.81E+18	687 ms	7,829	0	8.81E+18	670 ms	6,795	0
	ytopt	crashes due to Bayesian search				crashes due to Bayesian search			
	KTT	4.89E+18	1293 ms	25,367	0	4.89E+18	1303 ms	18,233	0
	HyperMapper	crashes due to Bayesian search				crashes due to Bayesian search			
	Kernel Tuner	cannot express global size				cannot express global size			
	CLTune	2.32E+09	> 4.0h	0	0	2.32E+09	> 4.0h	0	0
	OpenTuner	2.00E+43	791 ms	0	210,090	2.00E+43	867 ms	0	195,182
CONV	**pyATF**	2.03E+08	43433 ms	10,874	0	2.03E+08	46866 ms	18,473	0
	BaCO	crashes due to Bayesian search				crashes due to Bayesian search			
	ATF	2.03E+08	4290 ms	11,885	0	2.03E+08	1999 ms	21,586	0
	ytopt	crashes due to Bayesian search				crashes due to Bayesian search			
	KTT	2.03E+08	9477 ms	26,302	0	2.03E+08	14932 ms	34,412	0
	HyperMapper	crashes due to Bayesian search				crashes due to Bayesian search			
	Kernel Tuner	cannot express global size				cannot express global size			
	CLTune	0	80105 ms	0	0	0	143508 ms	0	0
	OpenTuner	8.59E+36	796 ms	0	202,064	8.59E+36	873 ms	0	187,446
PRL	**pyATF**	2.31E+07	1057 ms	354	0	2.31E+07	1091 ms	195	0
	BaCO	crashes due to Bayesian search				crashes due to Bayesian search			
	ATF	2.31E+07	322 ms	261	0	2.31E+07	317 ms	271	0
	ytopt	crashes due to Bayesian search				crashes due to Bayesian search			
	KTT	8.46E+06	1369 ms	101	0	8.46E+06	1378 ms	101	0
	HyperMapper	crashes due to Bayesian search				crashes due to Bayesian search			
	Kernel Tuner	cannot express global size				cannot express global size			
	CLTune	custom data-types not supported				custom data-types not supported			
	OpenTuner	1.74E+26	882 ms	0	200,881	1.74E+26	873 ms	0	182,118
GEMM	**pyATF**	1.08E+08	1885 ms	10,709	0	1.08E+08	1896 ms	15,614	0
	BaCO	crashes due to Bayesian search				crashes due to Bayesian search			
	ATF	1.08E+08	341 ms	10,805	0	1.08E+08	345 ms	16,108	0
	ytopt	crashes due to Bayesian search				crashes due to Bayesian search			
	KTT	3.90E+07	2448 ms	34,737	0	3.90E+07	2499 ms	17,366	0
	HyperMapper	crashes due to Bayesian search				crashes due to Bayesian search			
	Kernel Tuner	cannot express global size				cannot express global size			
	CLTune	1,008	22 min	1,008	0	1,008	23 min	1,008	0
	OpenTuner	1.71E+28	840 ms	0	196,618	1.71E+28	836 ms	0	181,289

Figure 3. Characteristics of auto-tuning runs reported in Figure 2

tools, because it cannot be configured to use warm-up runs and multiple evaluations of a configuration – to achieve high-quality measurements, we use 3 warm-up runs and take the average of 5 evaluations for all other tools (including pyATF). Note that pyATF, ATF, and KTT generate and store the constrained search space in order to explore valid configurations only. In contrast, OpenTuner keeps invalid configurations within its space, which significantly simplifies OpenTuner's implementation [61]. However, keeping invalid configurations within the space severely limits OpenTuner's auto-tuning capabilities for recent parallel implementations whose spaces contain many invalid configurations when constraints on tuning parameters are ignored (see Table 1), as discussed in the following.

In Figure 2, we observe that pyATF, even though based on Python, achieves for our four application case studies, on GPU and CPU, analogous high quality auto-tuning results as the state-of-the-art ATF tool which is implemented in the performance-oriented C++ programming language [61]. This is because the Python overhead (e.g., of ∼ 40 seconds for example CONV) is amortized during the overall auto-tuning process which runs 4 hours in Figure 2.

Tools BaCO, ytopt, and HyperMapper rely on the Bayesian search which has difficulties with exploring the search spaces of our case studies: our studies rely on complex search spaces that are high-dimensional to achieve high performance on different kinds of architectures and for different data characteristics (often higher than hand-optimized vendor libraries, as reported in Figure 2). The Bayesian search crashes for exploring these spaces because of their high dimensionality.[9]

The KTT tool often struggles with achieving the same auto-tuning efficiency as other tools. This is because KTT uses random search to explore its search spaces, which is particularly inefficient for large spaces (e.g., the space of CCSD(T)). In contrast, pyATF implements and uses the efficient AUC search technique.

Kernel Tuner has difficulties with expressing the OpenCL global sizes of our studies and thus cannot be used for auto-tuning: Kernel Tuner is limited to expressing global sizes that consist of constants, optionally divided by the value of a tuning parameter. However, the global sizes of our studies are complex arithmetic expressions of constants and tuning parameters (e.g., containing also multiplications) to achieve high performance [53]. For such complex requirements, pyATF allows arbitrary Python callables as global and local sizes. Note that even when Kernel Tuner would be extended to support expressing global sizes as required by our case studies, it would suffer from similar issues as CLTune, because it internally relies on the same search space generation and storing techniques as CLTune (instead of on the techniques implemented in pyATF [61]).

CLTune struggles with achieving the auto-tuning results of pyATF, because we have to use hand-pruned parameter ranges for CLTune (as discussed above) to enable CLTune generating its spaces in acceptable time. However, hand-pruning parameter ranges usually misses well-performing configurations, because well-performing configurations often differ significantly for different architectures and data characteristics [61]. Furthermore, requiring hand-pruning reduces the usability of CLTune, because pruning requires considerable expert knowledge from the user, who generally has only limited knowledge of hardware and optimization details.

[9]Recent research on Bayesian-based optimizations (e.g., by Hvarfner et al. [31]) aims to widen the applicability of Bayesian search, but these research insights have not been implemented in auto-tuning tools yet. Note that Hellsten et al. [29] experimentally compare BaCO against ATF, based on case studies relying on simpler, lower-dimensional search spaces than the spaces considered in this work.

OpenTuner cannot find valid configurations within its spaces, because its spaces contain invalid configurations: for our studies, only a fraction of $< 0.00001\%$ represent valid configurations within the spaces of OpenTuner (as reported in Table 1). Consequently, OpenTuner was not able to find a single valid configuration within the 4h of tuning time, for any case study.

4.2 Further Case Studies

In the previous subsection, we have experimentally shown that pyATF – even though offering an easy-to-use, Python-based user interface (introduced and discussed in Section 3) – achieves auto-tuning results of the same high quality as the ATF tool, because pyATF internally implements the same auto-tuning techniques as ATF [61].

Previous work has successfully used ATF for auto-tuning important real-world case studies from popular domains, summarized in Figure 4. Consequently, we argue that pyATF is likely to also achieve the same high-quality auto-tuning results for the case studies in Figure 4 as ATF, because both pyATF and ATF internally implement the same auto-tuning concepts.

	Domains	Applications
1	Compiler Optimization	GCC Flags [56], SIMD Vectorization [69]
2	Data Mining	Probabilistic Record Linkage [59]
3	Quantum Chemistry	CCSD(T) [53]
3	Deep Learning	BLAS [53, 56]
4	Sparse Computations	SpMM, SpMV, TTV [29]
5	DSL Compiler Optimizations	Lift [26, 37], MDH [53, 55, 58], RISE [29]
6	Polyhedral Compilation	PPCG, PluTo [26, 53]
7	Signal Processing	FFT [37]
8	Stencil Computations	Conv [53, 56, 58], Jacobi2D/3D, ... [26, 53, 58, 68]
9	FPGA Programming	BFS, Audio, PreEuler [29]

Figure 4. Applications auto-tuned using ATF

5 Related Work

Auto-tuning is a popular technique in the area of compiler construction. Many existing approaches achieve high auto-tuning efficiency [4, 5, 11–15, 17, 19, 21–23, 25, 32, 33, 35, 36, 38–41, 43, 44, 51, 63, 64, 66, 67, 70–72, 76, 77, 81]. However, these approaches are often limited in their applicability, by being specifically designed and optimized for a particular target domain only (e.g., only linear algebra [77]), thus also known as *non-generic* auto-tuners.

Our work is inspired by *generic* auto-tuning approaches which widen the applicability of auto-tuning by targeting arbitrary application domains. The state-of-the-art generic auto-tuning tools are summarized in Figure 1. These tools often rely on proof-of-concept user interfaces, because the tools are generally focused on presenting and demonstrating scientific contributions instead of being useful tools in practice. Moreover, many of the existing auto-tuning tools struggle with efficiently handling tuning parameters that are constrained (see Figure 1), which limits their auto-tuning

efficiency for modern parallel implementations (as experimentally confirmed in Section 4). Also, the existing tools are often not offered in the increasingly popular Python programming language and thus complex to extend (e.g., by new search techniques) and integrate into Python-based projects (e.g., modern deep learning frameworks and tools [3, 42, 49]). Some auto-tuning tools are also limited to auto-tuning programs in certain programming languages only (Figure 1).

The ATF approach is most close to pyATF, but its user interface is based on a DSL (Domain Specific Language) [56, 61] or C++ [57], which both were experienced and reported by real-world ATF users as notably more complex to use than our new Python-based interface for pyATF.

The pyATF auto-tuning tool introduced in this paper aims to improve the usability, functionality, and/or applicability of the existing generic auto-tuning tools in Figure 1. The user interface of pyATF is designed with a particular focus on usability for real-world users. For example, apart from the facts that pyATF works immediately out-of-the-box, can be conveniently installed via the Python pip package manager, is extensively documented and illustrated on a website, and is publicly available and open source, we discuss in Section 3 that pyATF significantly improves each step required in generic auto-tuning (discussed in Section 3): *1. Generating the Search Space:* by allowing arbitrary Python callables as constraint function of parameters, and by offering expressive range types for parameters; *2. Implementing a Cost Function:* by targeting programs in arbitrary languages to tune, and by offering pre-implemented cost functions (e.g., for CUDA and OpenCL programs); *3. Exploring the Search Space:* by offering a rich set of pre-implemented, well-proven numerical search techniques, including so-called meta-techniques that automatically choose the technique for the user, and by offering a wide set of abort conditions that can be arbitrarily combined by the user to meet complex, real-world user requirements – pyATF is designed generically such that new search techniques and abort conditions can be easily added to pyATF for special user requirements. Particularly, pyATF is designed and offered in the Python programming language which is experienced as productive [1], allows easily extending pyATF (e.g., by new search techniques and abort conditions), and also allows easy integration into Python-based projects, e.g., modern deep learning frameworks and tools, such as TensorFlow [3], PyTorch [49], and the recent Mojo programming approach [42]. In Section 4, we confirm experimentally that even though pyATF offers an expressive, easy-to-use, Python-based user interface, it achieves auto-tuning results of the same (or even higher) quality as the state-of-the-art auto-tuning tools in Figure 1.

More classic generic auto-tuning tools include *Active Harmony* [74] and *Orio* [27] which both struggle with efficiently handling constrained tuning parameters [61] – a severe limitation when auto-tuning recent parallel implementations.

Other kinds of approaches are focused on generating auto-tunable code, rather than being auto-tuning tools themselves, e.g., TVM [16] and MDH [53, 58, 60] (both used for code generation in Section 4).

6 Conclusion

We introduce the generic auto-tuning tool pyATF which introduces an easy-to-use, expressive user interface for auto-tuning and that combines three major advantages of the existing state-of-the-art generic auto-tuning tools: i) pyATF's user interface is offered in the Python programming language which is becoming increasingly popular, particularly among non-expert users, and it allows easily extending pyATF (e.g., by new search techniques and abort conditions) and integrating it into Python-based projects, e.g., modern deep learning frameworks and tools; ii) pyATF supports constrained tuning parameters, as required for efficiently auto-tuning modern parallel implementations; iii) pyATF can be used for auto-tuning programs written in arbitrary programming languages. We argue that pyATF's Python-based user interface is easier to use and more expressive than the current state-of-the-art auto-tuning user interfaces (e.g., regarding expressing constraint functions and defining abort conditions) – improving the usability of auto-tuning tools has been identified as an ongoing, major challenge in auto-tuning [9, 10]. Moreover, pyATF works immediately out-of-the-box, is publicly available and open source, can be conveniently installed via the Python pip package manager, and is extensively documented and illustrated on a website[10]. Our experiments confirm that even though pyATF is based on a user-friendly Python interface, it achieves auto-tuning results of the same (or even higher) quality than state-of-the-art auto-tuning tools, for real-world case studies from important areas: quantum chemistry, image processing, data mining, and deep learning. We also argue that pyATF can be used in many further popular domains, including compiler optimizations and FPGA programming.

We consider pyATF as a useful tool to assist research and industrial projects focused on compiler construction, due to pyATF's usability, broad applicability, and performance achievements for real-world case studies.

Acknowledgment

This work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – project *PPP-DL (470527619)*. We thank Johannes Lenfers from the BaCO team for his helpful comments and proofreading this paper.

[10] https://atf-tuner.org

References

[1] 2023. TIOBE – The Software Quality Company. https://www.tiobe.com/tiobe-index/.

[2] 2024. KTT - Kernel Tuning Toolkit. https://github.com/HiPerCoRe/KTT/tree/master.

[3] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 265–283. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi

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

[5] Willow Ahrens, Fredrik Kjolstad, and Saman Amarasinghe. 2022. Autoscheduling for sparse tensor algebra with an asymptotic cost model. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 269–285. https://doi.org/10.1145/3519939.3523442

[6] 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* (Edmonton, AB, Canada) *(PACT '14)*. Association for Computing Machinery, New York, NY, USA, 303–316. https://doi.org/10.1145/2628071.2628092

[7] Apache TVM Community. 2022. Invalid comm_reducer. https://discuss.tvm.apache.org/t/invalid-comm-reducer/12788.

[8] Artifact Implementation. 2025. https://doi.org/10.1145/3580444.

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

[10] Protonu Basu, Mary Hall, Malik Khan, Suchit Maindola, Saurav Muralidharan, Shreyas Ramalingam, Axel Rivera, Manu Shantharam, and Anand Venkat. 2013. Towards making autotuning mainstream. *The International Journal of High Performance Computing Applications* 27, 4 (2013), 379–393. https://doi.org/10.1177/1094342013493644

[11] 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.

[12] 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)*. IEEE, 307–316.

[13] James Bergstra, Brent Komer, Chris Eliasmith, Dan Yamins, and David D Cox. 2015. Hyperopt: a Python library for model selection and hyperparameter optimization. *Computational Science & Discovery* 8, 1 (jul 2015), 014008. https://doi.org/10.1088/1749-4699/8/1/014008

[14] 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. *Microprocessors and Microsystems* 37, 8, Part C (2013), 1073 – 1089. https://doi.org/10.1016/j.micpro.2013.06.001 Special Issue on European Projects in Embedded System Design: EPESD2012.

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

[16] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 578–594. https://www.usenix.org/conference/osdi18/presentation/chen

[17] 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.

[18] T Daniel Crawford and Henry F Schaefer III. 2007. An introduction to coupled cluster theory for computational chemists. *Reviews in computational chemistry* 14 (2007), 33–136.

[19] Simon Garcia De Gonzalo, Simon D. Hammond, Christian R. Trott, and Wen-Mei Hwu. 2017. Revisiting Online Autotuning for Sparse-Matrix Vector Multiplication Kernels on Next-Generation Architectures. 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)*. 72–80. https://doi.org/10.1109/HPCC-SmartCity-DSS.2017.10

[20] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*. 248–255. https://doi.org/10.1109/CVPR.2009.5206848

[21] Christophe Dubach, John Cavazos, Björn Franke, Grigori Fursin, Michael FP 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.

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

[23] 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 Compiler". *International journal of parallel programming* 39, 3 (2011), 296–327.

[24] GitHub. 2024. NumExpr: Fast numerical expression evaluator for NumPy. https://github.com/pydata/numexpr.

[25] GitHub. 2024. Scikit-Optimize. https://github.com/scikit-optimize/scikit-optimize.

[26] 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* (Vienna, Austria) *(CGO 2018)*. ACM, New York, NY, USA, 100–112. https://doi.org/10.1145/3168824

[27] 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.

[28] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. *CoRR* abs/1512.03385 (2015). arXiv:1512.03385 http://arxiv.org/abs/1512.03385

[29] Erik Orm Hellsten, Artur Souza, Johannes Lenfers, Rubens Lacouture, Olivia Hsu, Adel Ejjeh, Fredrik Kjolstad, Michel Steuwer, Kunle Olukotun, and Luigi Nardi. 2024. BaCO: A Fast and Portable Bayesian Compiler Optimization Framework. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4* (Vancouver, BC, Canada) *(ASPLOS '23)*. Association for Computing Machinery, New York, NY, USA, 19–42. https://doi.org/10.1145/3623278.3624770

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

[31] Carl Hvarfner, Erik Orm Hellsten, and Luigi Nardi. 2024. Vanilla Bayesian Optimization Performs Great in High Dimensions. arXiv:2402.02229 [cs.LG]

[32] 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)*. 254–261.

[33] 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)*. 387–400.

[34] Khronos. 2023. The OpenCL Specification. https://registry.khronos.org/OpenCL/specs/3.0-unified/html/OpenCL_API.html.

[35] A. E. Kiasari, Z. Lu, and A. Jantsch. 2013. An Analytical Latency Model for Networks-on-Chip. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 21, 1 (2013), 113–123.

[36] 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* (London, United Kingdom) *(KDD '18)*. Association for Computing Machinery, New York, NY, USA, 443–452. https://doi.org/10.1145/3219819.3219837

[37] 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* (Berlin, Germany) *(FHPNC 2019)*. ACM, New York, NY, USA, 1–13. https://doi.org/10.1145/3331553.3342613

[38] 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* (Washington DC, USA) *(PLDI '04)*. Association for Computing Machinery, New York, NY, USA, 171–182. https://doi.org/10.1145/996841.996863

[39] 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* (2012).

[40] Richard Liaw, Eric Liang, Robert Nishihara, Philipp Moritz, Joseph E. Gonzalez, and Ion Stoica. 2018. Tune: A Research Platform for Distributed Model Selection and Training. arXiv:1807.05118 [cs.LG] https://arxiv.org/abs/1807.05118

[41] 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* (Houston, Texas, USA) *(GPGPU-6)*. Association for Computing Machinery, New York, NY, USA, 66–75. https://doi.org/10.1145/2458523.2458530

[42] Modular. 2024. Mojo – the programming language for all AI developers. https://www.modular.com/max/mojo.

[43] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. 2015. PolyMage: Automatic Optimization for Image Processing Pipelines. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (Istanbul, Turkey) *(ASPLOS '15)*. Association for Computing Machinery, New York, NY, USA, 429–443. https://doi.org/10.1145/2694344.2694364

[44] 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.

[45] Luigi Nardi, David Koeplinger, and Kunle Olukotun. 2019. Practical Design Space Exploration. In *2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. 347–358. https://doi.org/10.1109/MASCOTS.2019.00045

[46] Cedric Nugteren. 2018. CLBlast: A Tuned OpenCL BLAS Library. In *Proceedings of the International Workshop on OpenCL* (Oxford, United Kingdom) *(IWOCL '18)*. Association for Computing Machinery, New York, NY, USA, Article 5, 10 pages. https://doi.org/10.1145/3204919.3204924

[47] 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*. 195–202. https://doi.org/10.1109/MCSoC.2015.10

[48] NVIDIA. 2023. NVRTC. https://docs.nvidia.com/cuda/nvrtc/index.html.

[49] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.), Vol. 32. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf

[50] Filip Petrovič, David Střelák, Jana Hozzová, Jaroslav Ol'ha, Richard Trembecký, Siegfried Benkner, and Jiří Filipovič. 2020. A benchmark set of highly-efficient CUDA and OpenCL kernels and its dynamic autotuning with Kernel Tuning Toolkit. *Future Generation Computer Systems* 108 (2020), 161–177. https://doi.org/10.1016/j.future.2020.02.069

[51] Markus Puschel, José MF 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.

[52] Python. 2024. `datetime` — Basic date and time types. https://docs.python.org/3/library/datetime.html.

[53] Ari Rasch. 2024. (De/Re)-Composition of Data-Parallel Computations via Multi-Dimensional Homomorphisms. *ACM Trans. Program. Lang. Syst.* (may 2024). https://doi.org/10.1145/3665643

[54] Ari Rasch, Julian Bigge, Martin Wrodarczyk, Richard Schulze, and Sergei Gorlatch. 2020. dOCAL: High-Level Distributed Programming with OpenCL and CUDA. *The Journal of Supercomputing* 76, 7 (2020), 5117–5138. https://doi.org/10.1007/s11227-019-02829-2

[55] Ari Rasch and Sergei Gorlatch. 2018. Multi-dimensional Homomorphisms and Their Implementation in OpenCL. *International Journal of Parallel Programming* 46, 1 (01 Feb 2018), 101–119. https://doi.org/10.1007/s10766-017-0508-z

[56] Ari Rasch and Sergei Gorlatch. 2019. ATF: A generic directive-based auto-tuning framework. *Concurrency and Computation: Practice and Experience* 31, 5 (2019), e4423. https://doi.org/10.1002/cpe.4423 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.4423 e4423 cpe.4423.

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

[58] Ari Rasch, Richard Schulze, and Sergei Gorlatch. 2019. Generating Portable High-Performance Code via Multi-Dimensional Homomorphisms. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 354–369. https://doi.org/10.1109/PACT.2019.00035

[59] 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* (Limassol, Cyprus) *(SAC '19)*. Association for Computing Machinery, New York, NY, USA, 526–533. https://doi.org/10.1145/3297280.3297330

[60] Ari Rasch, Richard Schulze, Denys Shabalin, Anne Elster, Sergei Gorlatch, and Mary Hall. 2023. (De/Re)-Compositions Expressed Systematically via MDH-Based Schedules. In *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction* (Montréal, QC, Canada) *(CC 2023)*. Association for Computing Machinery, New York, NY, USA, 61–72. https://doi.org/10.1145/3578360.3580269

[61] 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 (jan 2021), 26 pages. https://doi.org/10.1145/3427093

[62] Ari Rasch, Martin Wrodarczyk, Richard Schulze, and Sergei Gorlatch. 2018. OCAL: An Abstraction for Host-Code Programming with OpenCL and CUDA. In *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*. 408–416. https://doi.org/10.1109/PADSW.2018.8644541

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

[64] 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.

[65] Paul Springer and Paolo Bientinesi. 2016. Design of a high-performance GEMM-like Tensor-Tensor Multiplication. *CoRR* (2016). arXiv:1607.00145 [quant-ph] http://arxiv.org/abs/1607.00145

[66] 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

[67] 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 Heidelberg, Berlin, Heidelberg, 72–83.

[68] 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, Article 52 (Dec. 2019), 25 pages. https://doi.org/10.1145/3368858

[69] 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* (Phoenix, Arizona) *(ICS '19)*. ACM, New York, NY, USA, 319–329. https://doi.org/10.1145/3330345.3331059

[70] 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)*. 649–660. https://doi.org/10.1109/HPCA.2017.14

[71] Thiago SFX Teixeira, William Gropp, and David Padua. 2019. Managing code transformations for better performance portability. *The International Journal of High Performance Computing Applications* 33, 6 (2019), 1290–1306.

[72] 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* (Washington, DC, USA) *(CGO 2019)*. IEEE Press, Piscataway, NJ, USA, 217–228.

[73] 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* (Denver, Colorado) *(SC '17)*. Association for Computing Machinery, New York, NY, USA, Article 43, 12 pages. https://doi.org/10.1145/3126908.3126939

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

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

[76] 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)*. 1–14.

[77] R.C. Whaley and J.J. Dongarra. 1998. Automatically Tuned Linear Algebra Software. In *SC '98: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*. 38–38. https://doi.org/10.1109/SC.1998.10004

[78] Stephen Wright and Jorge Nocedal. 1999. Numerical Optimization. *Springer Science* 35, 67–68 (1999), 7.

[79] Xingfu Wu, Prasanna Balaprakash, Michael Kruse, Jaehoon Koo, Brice Videau, Paul Hovland, Valerie Taylor, Brad Geltz, Siddhartha Jana, and Mary Hall. 2023. ytopt: Autotuning Scientific Applications for Energy Efficiency at Large Scales. arXiv:2303.16245 [cs.DC]

[80] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. 2020. Ansor: Generating High-Performance Tensor Programs for Deep Learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 863–879. https://www.usenix.org/conference/osdi20/presentation/zheng

[81] 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* (Limassol, Cyprus) *(PACT '18)*. Association for Computing Machinery, New York, NY, USA, Article 1, 12 pages. https://doi.org/10.1145/3243176.3243187

Received 2024-11-04; accepted 2024-12-21