BaCO: A Fast and Portable Bayesian Compiler Optimization Framework

Erik Hellsten
Lund University, Sweden
erik.hellsten@cs.lth.se

Artur Souza
Federal University of Minas Gerais, Brazil
arturluis@dcc.ufmg.br

Johannes Lenfers
University of Münster, Germany
j.lenfers@uni-muenster.de

Rubens Lacouture
Stanford University, USA
rubensl@stanford.edu

Olivia Hsu
Stanford University, USA
owhsu@stanford.edu

Adel Ejjeh
University of Illinois at Urbana-Champaign, USA
aejjeh@illinois.edu

Fredrik Kjolstad
Stanford University, USA
kjolstad@stanford.edu

Michel Steuwer
TU Berlin, Germany
University of Edinburgh, UK
michel.steuwer@tu-berlin.de

Kunle Olukotun
Stanford University, USA
kunle@stanford.edu

Luigi Nardi
Lund University, Sweden
Stanford University, USA
luigi.nardi@cs.lth.se

ABSTRACT

We introduce the Bayesian Compiler Optimization framework (BaCO), a general purpose autotuner for modern compilers targeting CPUs, GPUs, and FPGAs. BaCO provides the flexibility needed to handle the requirements of modern autotuning tasks. Particularly, it deals with permutation, ordered, and continuous parameter types along with both known and unknown parameter constraints. To reason about these parameter types and efficiently deliver high-quality code, BaCO uses Bayesian optimization algorithms specialized towards the autotuning domain. We demonstrate BaCO's effectiveness on three modern compiler systems: TACO, RISE & ELEVATE, and HPVM2FPGA for CPUs, GPUs, and FPGAs respectively. For these domains, BaCO outperforms current state-of-the-art autotuners by delivering on average 1.36×–1.56× faster code with a tiny search budget, and BaCO is able to reach expert-level performance 2.9×–3.9× faster.

CCS CONCEPTS

• **Theory of computation** → **Mathematical optimization**; • **General and reference** → **Performance**.

KEYWORDS

Compiler optimizations, high-performance computing, Bayesian Optimization, Autotuning, Autoscheduling

ASPLOS '23, March 25–29, 2023, Vancouver, BC, Canada

ACM ISBN 979-8-4007-0394-2/23/03.
https://doi.org/10.1145/3623278.3624770

ACM Reference Format:
Erik Hellsten, Artur Souza, Johannes Lenfers, Rubens Lacouture, Olivia Hsu, Adel Ejjeh, Fredrik Kjolstad, Michel Steuwer, Kunle Olukotun, and Luigi Nardi. 2023. BaCO: A Fast and Portable Bayesian Compiler Optimization Framework . In *28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 24 pages. https://doi.org/10.1145/3623278.3624770

1 INTRODUCTION

Modern compilers are rapidly evolving to keep pace with the growing range of increasingly specialized hardware targets, as well as the ever-changing domains of interest. A recent trend is to separate policy (what to compute) from mechanism (transformations and code generation describing how to compute) by using scheduling languages. Prominent examples of this paradigm include Halide [44], TVM [7], TACO [49], and RISE & ELEVATE [19, 55]. This design pushes the optimization task of finding good schedules outside of the compiler core, where it can be done manually or automatically by an autoscheduler. Scheduling languages may express more complex optimization spaces, and, thus, require more advanced autotuning features to effectively and efficiently tackle the autoscheduling task. Modern hardware backends—like GPUs, as in RISE & ELEVATE [55], and FPGAs, as in HPVM2FPGA [13, 33]—further increase the complexity of relevant optimization spaces.

The separation of concerns between policy and mechanism in compilers exposes a great opportunity. If we can design a portable autoscheduler that is effective across many compilers, like the design shown in Fig. 1, then we can reduce the complexity of the overall ecosystem. New compilers get an autoscheduler with minimal effort, and improvements in the autoscheduler automatically benefit all compilers and their subsequent domains and backends.

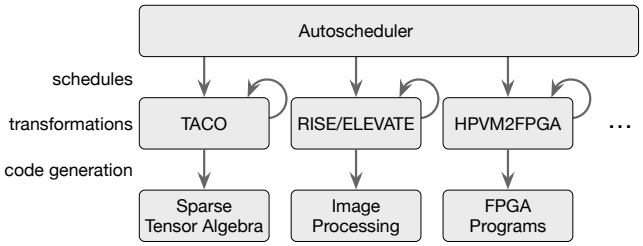

Figure 1: An autoscheduler that is portable across the scheduling languages of diverse compilers.

However, a portable autoscheduler must be designed with a rich input language to allow users to accurately describe the search space exposed by their particular compiler. This autoscheduling search space is determined by the product of the hardware target, the compiler's scheduling language features and configuration tuning parameters. In modern compilers, this search space is often complex, including both continuous parameters (e.g., real-valued tuning parameters), and discontinuous parameters broken down into integers (e.g., tiling factors), permutation categories (e.g., loop reordering), ordinals/ordered categories (e.g., unroll factors), and categoricals/unordered categories (parallelization schemes). These parameter types are often abbreviated to RIPOC [2, 22, 41].

However, even the large class of search spaces that can be generated as the Cartesian combination of these parameters is often inadequate. Scheduling parameters frequently depend on the settings of other parameters, leading to constraints on the scheduling space. One such example is a loop bound that must be an exact multiple of a given tiling factor. We refer to these as *known constraints*, which are provided to the autoscheduler ahead of time. Other constraints are initially unknown and must be learned throughout the autoscheduling. An example of this is learning sets of parameters that would generate programs that adhere to hardware constraints, such as avoiding out-of-memory errors on a GPU. Such constraints are often referred to as *hidden constraints*. For a general autotuning framework to be efficient and portable across a multitude of compilers, it needs to support as many of these features as possible.

Once able to express a complex search space, the autoscheduler must be effective and efficient in finding a good schedule within this space. For optimizing compilers, the performance of the generated code is of primary concern. Therefore, the autoscheduler will invariably use some sort of search combined with a cost model to evaluate points of the search space. The cost model could be analytic or data-driven, but the most accurate cost model is to generate and run the code on its target platform. For a general autoscheduler used across diverse compilers, a cost model based on running the code makes it easy to use the autoscheduler with a new compiler. We refer to empirical autoschedulers, whose cost model is to run the actual code generated by the compiler, as *autotuners*. To achieve composability, and to work effectively and easily across a diverse set of compilers, it is vital for an autoscheduler to treat each compiler as a black-box system. The autoscheduler's job will then be to optimize the black-box system using the smallest possible budget of trials and errors, i.e., evaluations of the black-box system.

	Parameters		Constraints	
	RIOC	Perm.	Hidden	Known
ATF [46]	✓	✗	✗	✓
OpenTuner [2]	✓	✓	✗	✗
Ytopt [63]	✓	✗	✗	✓
Kernel Tuner [61]	✓	✗	✗	✓
KTT [43]	✗	✗	✗	✓
GPTune [38]	✓	✗	✗	✓
HyperMapper [41]	✓	✗	✓	✗
Bliss [48]	✗	✗	✗	✗
DeepHyper [11]	✓	✗	✗	✓*
SMAC3 [36]	✓	✗	✗	✓*
GpyOpt [3]	✗	✗	✗	✓
Spearmint [52]	✓	✗	✓	✗
GPflowOpt [28]	✗	✗	✓	✗
cBO [17]	✗	✗	✓	✗
BaCO (ours)	✓	✓	✓	✓

Table 1: Framework capabilities; RIOC abbreviates Real/Integer/Ordinal/Categorical parameters. Limited support for constraints is marked with ✓*.

Many successful autotuning frameworks have been proposed, some of which are listed in Table 1. These frameworks have helped deliver high-performance software in the past. However, they do not support all features required to effectively search over the complex search spaces described by diverse scheduling languages across modern domain-specific compilers targeting various hardware backends. For example, Table 2 shows the features required by three modern compilers (TACO, RISE & ELEVATE, and HPVM2FPGA), and Table 1 shows that none of 14 recently proposed autotuners support all required features in the manner we define below. Some of the frameworks in Table 1 do support some types of user-defined constraints. Several frameworks are using ConfigSpace [35], which supports conjunctions of linear constraints. While this is certainly useful, it is inadequate for defining non-linear dependencies (see frameworks marked with ✓*). The hidden constraints column in Table 1 means that the framework uses specialized tools to handle the constraint validity instead of assigning a high objective value to infeasible configurations, such as OpenTuner. Handling hidden constraints with a specialized tool distinguishes slow but valid configurations from invalid configurations, providing more information to the search mechanism. Lastly, while permutations can be cast as categoricals, BaCO is the first autotuning framework to make use of their underlying structure.

Hence, we propose *BaCO*, a novel general autotuning framework optimized towards the autotuning of modern compilers, which efficiently handles all features mentioned above. *BaCO* does not require any user-provided cost model but instead learns from observations from running the generated code throughout the optimization procedure. The support for sophisticated search spaces and online learning means that *BaCO* finds good schedules in fewer iterations than existing autotuners, while being easy to use. Notably, we do not adapt *BaCO* to individual compilers nor applications but show

	Parameters		Constraints	
	RIOC	Perm.	Hidden	Known
TACO [26]	✓	✓	✓	✓
RISE & ELEVATE [19]	✓	✓	✓	✓
HPVM2FPGA [13]	✓		✓	

Table 2: Features needed by different compilers; RIOC abbreviates Real/Integer/Ordinal/Categorical parameters.

that it yields expert-level performance out of the box for a wide range of applications. Our contributions are:

- The first Bayesian autotuning framework that supports all RIPOC features, including permutation types, through the definition of separate distance metrics, thus improving search performance (Sec. 3 and Sec. 4).
- The integration of a feasibility model for hidden constraints, simplifying the portability to new compiler backends (Sec. 4).
- The first system to use the chain-of-tree technique in a Bayesian optimization setting for portable autotuning on sparse search spaces (Sec. 4).
- Applying autotuning to three distinct compiler frameworks for different domains and hardware targets, along with a survey of the autotuning challenges of those three recent domain-specific compilers (Sec. 2).

The end result is that *BaCO* generates expert-level code significantly faster than the state of the art. We demonstrate the effectiveness and robustness of *BaCO* across three real-world compilers and code generation systems targeting CPUs, GPUs, and FPGAs (Sec. 5). *BaCO* reliably produces good schedules across different compilers without any customization for each compiler, such as hyperparameter tuning or custom constraint filtering. It achieves expert performance on TACO 3.15×–5.0× faster than the state-of-the-art, while RISE & ELEVATE achieves 1.35×–1.58× better performance with a tiny autotuning budget, and HPVM2FPGA achieves peak performance 2.43×–2.77× faster.

2 COMPLEXITY OF MODERN AUTOTUNING

To develop the next generation general-purpose autotuning framework, we need to better understand the real-life challenges faced by various modern compiler frameworks. Therefore, we investigate the autotuning features needed by the TACO, RISE & ELEVATE, and HPVM2FPGA compiler frameworks. This will allow us to identify an autotuning framework that is able to generalize across a wide spectrum of compilers and backend targets. As we shall see, this ideal general-purpose autotuning framework needs to support the features described in Table 2, which includes support for a wide range of parameter types and both hidden and known constraints.

The Tensor Algebra Compiler (TACO).. TACO [26] is the state-of-the-art compiler for sparse tensor algebra. It generates high-performance code for tensor operations expressed in a high-level Einstein notation, such as the sampled dense-dense matrix multiplication (SDDMM) computation represented as $A(i,j) = B(i,j) * C(i,k) * D(k,j)$. A particular strength of TACO is its capability to generate code for a large variety of sparse tensor formats [8].

TACO's scheduling language defines an iteration space transformation framework that dictates how to traverse a tensor stored in any particular format [49]. This provides a way to introduce optimization transformations, such as tiling, parallelization, vectorization, loop reordering, and more. An autotuning framework selecting the optimizations exposed by the scheduling language needs to provide not just traditional real, integer, ordinals, and categorical parameters as provided by most frameworks in Table 1, but also *permutation* parameters for selecting loop reordering. Typically, optimization is performed inside the compiler and controlled by a heuristic, but in TACO, as well as other similar compilers with scheduling languages, it is exposed as a tunable parameter. These optimization parameters also need to follow known constraints that TACO provides. An example is loop reordering variables that TACO enforces for concordant traversal.

RISE & ELEVATE.. RISE [55] and ELEVATE [19] are a powerful combination of compiler and scheduling languages. Computations are described in the RISE [55] language using well-known data-parallel patterns like `map` and `reduce` in the spirit of LIFT [20, 56]. Optimizations are applied and described in the ELEVATE [19] scheduling language as compositions of semantic preserving rewrite-rules. The optimized RISE program is compiled to high-performance CPU or GPU code.

Transformations, such as loop tiling, may introduce numerical tuning parameters, such as a tile size, which are often constrained by other numerical values, such as loop bounds. When automatically optimizing RISE programs, an explorative rewrite process speculatively applying program transformations is performed. To evaluate the performance of a transformed program, the system relies on an autotuning framework to pick all numerical parameters while respecting all known parameter constraints that the system can collect automatically and provide to the autotuning framework. Compiling for GPUs also introduces hidden constraints for the autotuning framework, such as choosing only parameter values that will result in a program fitting in the tight register and memory requirements. When these constraints are not satisfied, the compiler generates code that will fail to execute. Therefore, the autotuning framework must be able to learn these hidden constraints automatically.

HPVM2FPGA.. HPVM2FPGA [13] is a compiler that enables hardware-agnostic programming of field-programmable gate arrays (FPGAs). The compiler uses sophisticated optimizations, coupled with design space exploration (DSE), to automatically tune and generate well-performing FPGA designs from programs that have not been written by hardware and FPGA experts. HPVM2FPGA is part of the Heterogeneous Parallel Virtual Machine (HPVM) compiler infrastructure [12, 33], which provides a retargetable virtual ISA and compiler IR for programming heterogeneous systems.

During HPVM2FPGA's DSE, compiler transformations, such as loop unrolling, greedy loop fusion, argument privatization, and kernel fusion are explored. HPVM2FPGA generates its parameter space automatically through a static analysis of the IR, and the design space varies depending on the size of the application being compiled. The majority of the parameters are boolean parameters, with hidden constraints among them, making it challenging to explore the space efficiently.

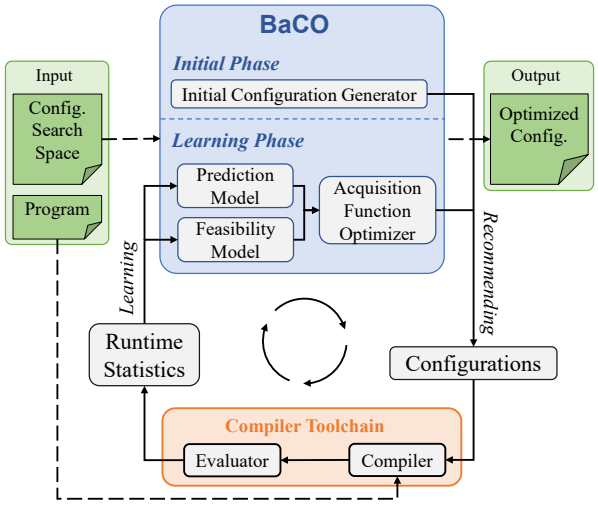

Figure 2: Overview of the BaCO framework.

3 THE BACO FRAMEWORK

We introduce *Bayesian Compiler Optimization* (BaCO)[1], a Bayesian optimization (BO) framework that learns high-performing auto-scheduling strategies. BaCO thrives in a small data world where configuration evaluations are costly, either due to high runtimes of the kernel or expensive simulations of code generation passes. BaCO is backend-agnostic, and it can be equally applied to CPU, GPU, and FPGA compilers. Building on the BO paradigm, BaCO is centred around a *configuration recommendation-evaluation* loop: it recommends promising new configurations that are subsequently scheduled and evaluated by the corresponding compiler toolchain. The evaluation results are used to fit two predictive models: one modelling the predicted value and one modelling the predicted probability of feasibility of new configurations. To initialize the two models, the procedure starts with an *initial phase*, where the first few configurations are sampled uniformly at random from the search space.

However, for BO to reach its full potential, it needs to be customized for autotuning tasks. This section explains the various modules of BaCO's architecture, shown in Fig. 2, whereas further specialization towards autotuning search spaces is emphasized in Sec. 4.

3.1 Bayesian optimization

Bayesian optimization is a steadily growing methodology for solving black-box optimization problems. Those are problems where the objective function $f(\boldsymbol{x})$ can only be accessed point-wise through expensive evaluations. At the core of BO is the use of a surrogate model, which estimates the objective function. This helps with performing well-informed decisions about which configurations to evaluate next. The goal is to find a good configuration in as few evaluations as possible. The most common choices of surrogate model are *Gaussian Process* (GP) or *Random Forest* (RF) predictors, which both provide useful uncertainty estimates. For any given point $\boldsymbol{x}$, the model provides mean and variance, which is used to balance exploration and exploitation. This trade-off is quantified by an acquisition function. Common examples are *Expected Improvement* and *Lower Confidence Bound*. The surrogate model is dynamically updated to learn from observations, creating a feedback loop where the BO framework proposes new points, that are then evaluated. The information from the evaluation is subsequently used to train the model. BO was historically developed for continuous compact domains, and the extension towards more exotic search spaces is currently being used in this work and studied by the BO community.

[1]Baco is Italian for bug.

3.2 Surrogate models over compiler domains

Choice of probabilistic model. One core element of an efficient BO algorithm is an accurate surrogate model [15]. While complex parameter domains have little impact on less intricate methods such as random sampling, the success of BO depends greatly on clever handling of such parameters. While traditionally, Random Forests have been considered the natural choice as surrogate models over discrete domains [4, 23, 41], recent studies show that a careful implementation of Gaussian Processes yields superior accuracy [9, 18]. However, to achieve the true potential of GPs in autotuning and DSE applications, significant customization of the GP is needed. This customization is explained in detail in the following sections. Sec. 5.3 shows the impact of some of these major design choices and a comparison between GPs and RFs.

GP kernel similarity function. A key feature in autotuning and DSE is the mixed-variable search space. Thus, the kernel needs to combine distance measures over different parameter types. We propose the weighted Euclidean norm $||\boldsymbol{d}||_2^2 = \sum_{i=1}^{D} (d_i/l_i)^2$ over the vector of individual distance measures $\boldsymbol{d}$, as a unified distance measure. D denotes the dimension of the search space, i.e., the number of parameters being optimized, and l_i are the horizontal length-scales, learned using maximum likelihood estimation (MLE) [40], weighting the different parameters. We use the 5/2-Matérn kernel [47], given by

$$k(\boldsymbol{x}, \boldsymbol{x}') = \sigma \left(1 + \sqrt{5}d + 5d^2\right) e^{-\sqrt{5}d}, \tag{1}$$

$$d = \sqrt{\sum_{i=1}^{D} \frac{d(x_i, x_i')^2}{l_i^2}} \tag{2}$$

where $d(x_i, x_i')$ denotes the distance between x_i and x_i' (described in Sec. 4.1), as this has shown to be efficient in many real life applications [27, 58]. To increase stability, we assume that the value observed in each evaluation, $y(\boldsymbol{x})$, is perturbed by some normally distributed noise [15], such that $y(\boldsymbol{x}) = f(\boldsymbol{x}) + \varepsilon$ and $\varepsilon \sim N(0, \sigma_\varepsilon)$.

GP hyperparameter optimization. A crucial element in effective optimization using GPs is to find good hyperparameters for the model. Especially important are the length-scales l_i presented in Eq. (2), which balance the importance ratio between different parameters. The remaining hyperparameters are the outputscale σ in Eq. (1), and the magnitude of the Gaussian noise σ_ε. BaCO optimizes the hyperparameters using a multistart gradient descent approach, which first uniformly samples a number of possible hyperparameter settings, then chooses a fraction of those with highest likelihood, and optimizes them individually using L-BFGS [37].

Discrete parameter spaces offer a number of practical challenges when fitting the GP model. One such challenge is that the model hyperparameter optimization method described above frequently prefers to give close to zero lengthscale values to some parameters. In practice, this means that configurations which take different values for those parameters have close to zero similarity, making the GP behave as a sparse model. This is undesirable as it reduces the model's expressive power. To address this artifact of GP modeling, as well as to stabilize the hyperparameter selection, BaCO uses *gamma priors* [40] for the lengthscales. These priors are chosen to be flexible while cutting out extreme hyperparameter settings. In practice, stabilizing the lengthscales means that different parameters are given more equal importance, preventing certain parameters from becoming too dominant or too insignificant due to model over-fitting. Gamma prior distributions are chosen as they have positive support, can be made reasonably concentrated and have long tails towards both zero and infinity. Other good alternatives with similar properties would be the log-normal or inverse-gamma distributions. By normalizing the input data, BaCO can use a single set of priors that works well for the majority of parameters. Note that, this is an artifact from that many parameters take identical values in discrete spaces, which rarely occurs when working with continuous parameters.

3.3 Acquisition function

The acquisition function quantifies the anticipated utility of evaluating a new point. We use the Expected Improvement (EI) acquisition function [25], which balances exploration and exploitation. Autotuning and DSE are characterized by both discrete search spaces and noisy function evaluations, in which case we observe that standard EI has a tendency to overly prioritize re-sampling points with good values. To avoid this unintended behavior, we propose a modified EI acquisition function which predicts the expected improvement of observing a noise free evaluation of the blackbox function. Computing the EI without considering the noise in the GP makes sampling repeated points less likely.

BaCO optimizes the acquisition function by multi-start local search. Initially, a large number of configurations are sampled at uniformly random, of which the best configurations are chosen as starting points for the local search. Neighbours are defined as all configurations that can be reached by modifying a single parameter.

4 ADAPTING TO EXOTIC SEARCH SPACES

When implementing an efficient autotuning framework, effectively handling all of the search space features is key. As BaCO is built around a GP predictive model, careful design of the distance metrics used for different variable types is of additional importance. In this section we study the intricacies of the different parameter types as well as how to handle known and hidden constraints.

4.1 Parameter types

Continuous, integer, and ordinal parameters. These types of parameters have the property that the values are comparable, i.e., you can use the greater or equal sign to order them. This can naturally and explicitly be translated into a distance metric, and in particular we use the absolute difference, $d(x_i, x_i') = |x_i - x_i'|$. However, certain such parameters are innately exponential in nature, such as tile size parameters. In that case, we use the Euclidean distance over a log-transformed space instead, $d(x_i, x_i') = |\log x_i - \log x_i'|$. The log transformation often more accurately describes the relationship between values. Consider tile sizes as an example. We expect the tile sizes 2 and 4 to be roughly as similar to each other as the tile sizes 512 and 1024. However, the tile sizes 512 and 514 would be much more similar than the pairs above.

Categorical parameters. Categorical parameters differ from ordinals in that they have no inherent order. Here, we use the *Hamming distance*, defined as $d_h(x_i, x_i') = \mathbb{1}_{x_i \neq x_i'}$, where $\mathbb{1}$ is the indicator function, which returns 1 if $x_i \neq x_i'$ and 0 otherwise. In other words, the Hamming distance only considers whether the parameter values are identical or not. The scale here is not relevant as the distance is weighted by the lengthscale l_i in Eq. (2).

Permutation parameters. Permutation parameters are used to describe the reordering of a sequence of elements. In compiler applications this most commonly appears as the reordering of a set of loops [22]. Consider for example a kernel with four nested loops (l_1, l_2, l_3, l_4) which can be performed in any order. This ordering can be represented by a single permutation variable $\boldsymbol{\pi}$, which is a vector whose element i, given by $\pi_i = j$, describes the index j of loop l_i in the new order. For example, the permutation $\boldsymbol{\pi} = [2, 4, 3, 1]$ corresponds to the following loop reordering:

```
for (l1 ... )                          for (l4 ... )
   for (l2 ... )                          for (l1 ... )
      for (l3 ... )            →             for (l3 ... )
         for (l4 ... )                          for (l2 ... )
            ...                                    ...
```

Prior black-box optimization literature, autotuning, and DSE frameworks lack the capability to effectively handle this variable type, with the notable exception of OpenTuner [2]. In BO frameworks employing GPs, it is important to accurately estimate how different permutations relate to each other. In other words, for the nested loop reordering example above, the framework needs to determine if the two different loop orderings are likely to yield a similar performance. One naive way of handling permutation variables is to treat them as categorical variables, e.g., to consider one nested loop ordering to be equally similar to every other loop ordering (with the exception of itself). This, however, ignores the underlying structure that can be used to define a more refined similarity measure. We instead present three different semimetrics for permutations: the *Kendall distance*, *Spearman's rank correlation*, and the *Hamming distance*. While the semimetrics are not strictly distance metrics, Lomelí *et al.* [39] show that they can be used to form a valid GP kernel. These three semimetrics are illustrated in Fig. 3 on a set of four elements, where the two boxes represent two permutations $\pi = [1, 2, 3, 4]$ and $\pi' = [2, 4, 3, 1]$.

The first semimetric is the Kendall distance,

$$d_k(\pi, \pi') = \sum_{i=1}^{m} \sum_{j=1}^{m} |\mathbb{1}_{\pi_i' > \pi_j} - \mathbb{1}_{\pi_i > \pi_j'}|.$$

The Kendall distance represents the number of discordant pairs, i.e., the elements that have swapped order between the two permutations. Each discordant pair is represented by a green, interconnected

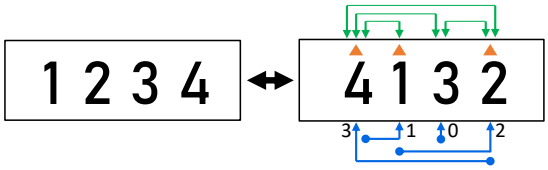

Figure 3: Illustration of similarity metrics between two permutations. The number of discordant pairs *(top of right-hand box, green)* is the Kendall distance, the squared element movement *(bottom of right-hand box, blue)* is the Spearman's rank correlation, and the number of changing elements *(triangles, orange)* is the Hamming distance.

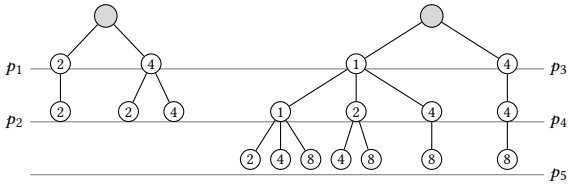

Figure 4: Chain-of-Trees with 5 parameters, p_1 to p_5.

double arrow. The second semimetric is the Spearman's rank correlation,

$$d_s(\pi, \pi') = \sum_{i=1}^{m} (\pi_i - \pi'_i)^2,$$

which is the sum of squared movements of the elements between two permutations. It is illustrated with blue arrows in Fig. 3, where the dots represent the starting points and the arrows the final position. For example, the number two starts in the second position in π (left) and moves to the fourth position in π' (right), meaning that it has travelled a distance of two. The Spearman's rank correlation then sums the squared displacement of all elements. Note that the square substantially emphasizes large rank changes. Lastly, the Hamming distance,

$$d_h(\pi, \pi') = \sum_{i=1}^{m} \mathbb{1}_{\pi_i \neq \pi'_i},$$

is the number of elements in π that are no longer at their original position in π' – represented with orange triangles in the figure.

For a given permutation set, the choice of semimetric depends on how those permutation parameters are expected to impact the performance metric. Intuitively, Kendall distance focuses more on parameter order, whereas Spearman's rank correlation emphasizes large movements of individual elements. The Hamming distance only considers the number of elements changed and ignores where they moved to. As an example, consider the two loop orders

```
for (12 ... )
    for (13 ... )
        for (11 ... )
            for (14 ... )
                ...
```

and

```
for (14 ... )
    for (13 ... )
        for (11 ... )
            for (12 ... ).
                ...
```

They have a high Spearman's rank correlation due to the large movement of the first and last element and relatively smaller Kendall and Hamming distances, which is intuitive given the compiler transformation that this represents. This is backed by our ablation analysis in Sec. 5.3, where we observe that using Spearman's rank correlation outperforms the other alternatives. By consequence, we use Spearman's rank correlation as a default setting for permutation variables in BaCO.

4.2 Parameter constraints

For an autotuning framework to be truly competitive in the complex world of modern autoscheduling, it is essential to effectively handle constraints in the parameter search space. Constraints can be divided into known constraints, which are known prior to optimization, and hidden constraints, which are only discovered during optimization. BaCO is designed to support both these constraints.

Known constraints. In autotuning applications, users often possess expert knowledge regarding parameter configurations that lead to inefficient or even infeasible schedules. Incorporating this knowledge into the autotuning framework leads to significant performance improvements. The improvement becomes even greater when the feasible set makes up a small fraction of all possible configurations, i.e., when the search space is sparse. BaCO handles known constraints during the acquisition function optimization, and proposes only feasible configurations. As such, the surrogate model trains exclusively on feasible points.

BaCO uses a *Chain of Trees* (CoT) data structure to deal with sparse search spaces, which was first presented by Rasch *et al.* [46]. The CoT computes all of the feasible configurations a-priori and stores them as a collection (or "chain") of trees. Each tree corresponds to a group of co-dependent parameters, and parameters in different trees are independent of one another. For each tree, each level of the tree corresponds to a single parameter and each node in that level corresponds to a possible value for that parameter. Each path from the root to a leaf then represents a partial configuration, and the tree is built so that only feasible configurations are included. Consider for example the following search space:

$$p_1 \in \{2, 4\},\ p_2 \in \{2, 4\},\ p_3 \in \{1, 4\},\ p_4 \in \{1, 2, 4\},\ p_5 \in \{2, 4, 8\}$$

$$p_1 \geq p_2,\ p_4 \geq p_3,\ p_5 \geq 2p_4$$

In this example, there are five input parameters and three constraints. Parameters p_1 and p_2 as well as parameters p_3, p_4, and p_5 are co-dependent. We represent them with the CoT shown in Fig. 4. As the parameters in different trees are independent, any combination of partial feasible configurations from the different trees yields a feasible configuration. For example, the leftmost path in the left tree combined with the rightmost path in the right tree yields the feasible configuration

$$(p_1, p_2, p_3, p_4, p_5) = (2, 2, 4, 4, 8).$$

The use of the Chain-of-trees in BaCO is threefold. First random sampling can be made directly from the CoT. Secondly, instead of evaluating the constraints explicitly, it is significantly more efficient to check whether the configuration belongs to the CoT. Thirdly, it allows working with highly sparse search spaces which are common in the autotuning domain. In such sparse spaces, operating directly on the original domain becomes infeasible.[2]

[2]In the results section, we have used domain expert knowledge to manually transform the search spaces to limit the sparsity. This allows a more interesting comparison with previous methods.

Working with constrained spaces is inherently biased since the optimization method presented in Rasch *et al.* [45] prioritizes different configurations depending on the structure of the CoT. Their approach is equivalent to random sampling a configuration from the CoT by starting at each root node and then iteratively choosing a child with uniform probability, which is biased towards configurations in less dense parts of the tree. This bias is furthermore dependent on the order in which the parameters appear in the tree, which is an undesirable feature. We propose an alternative approach where we instead sample uniformly from the leaves of the trees, which is bias-free, meaning that the random sampling will be performed uniformly on the all the configurations. We study the impact of the bias in Sec. 5.

Another source of sparsity comes from how parameters are defined prior to optimization. In autotuning and DSE applications, due to the nature of how computers store information, parameters are often constrained to take exponential values. Treating such parameters as integers leads to sparsity in the search spaces. BaCO instead applies the logarithmic transformation to such parameters. This transformation makes the search space significantly denser and yields more natural distances for the GP. These qualities improve performance, as we shall see in Sec. 5.

Hidden constraints. Requiring the complete feasible domain definition from the user would severely limit the autotuner's usability. Some constraints are either too complicated to describe analytically or unknown a-priori. Instead, BaCO supports the concept of hidden constraints, learned automatically during optimization. BaCO uses a Random Forest model to predict the probability of feasibility for each configuration. It then extends the EI acquisition function presented in Sec. 3.3, by multiplying the EI with the probability of feasibility [41]. This should be compared with the naive approach of assigning high objective values to infeasible configurations, which suffers from difficulties with setting an accurate penalty. Such penalty terms are further often detrimental for the statistical model.

However, the practical interaction between the acquisition function based on the GP model and the RF feasibility predictor is complex. There is a constant trade-off between the feasibility model wanting to select feasible points and the value predictor that seeks to explore the unexplored infeasible regions. If the surrogate model becomes excessively confident within the feasible region, this balance tends to be skewed. In which case the selection fails to reliably find feasible points. As a practical solution, we consider a minimum feasible limit ε_f and only consider configurations with probability of feasibility greater than ε_f for evaluation. By randomly sampling a new ε_f each iteration, with $p(\varepsilon_f = 0) > 0$, we asymptotically guarantee to not cut away any solutions by doing this.

5 EVALUATION

We validate the efficiency, effectiveness, and generalizability of BaCO. We first introduce the reference autotuning methods that we use as a baseline to evaluate the performance of BaCO, followed by the benchmarks from the three real-world frameworks presented in Sec. 2. We then show the performance results. For lack of space we show the extensive empirical results on all the frameworks and benchmarks in Appendix A. All experiments are run for 30 repetitions. We also show a wall-clock time analysis of all the autotuners used in Appendix B. The BaCO code is open-source and available at https://github.com/baco-authors/baco.

We answer the following research questions (RQ):

RQ1) Does BaCO achieve high performance with a limited autotuning budget? The evaluation in Fig. 5 shows that, with a tiny budget of 20–40 evaluations, depending on the complexity of the benchmark, *BaCO* achieves 1.35×–1.55× better performance than the state-of-the-art baselines. Furthermore, BaCO consistently achieves expert-level performance with a small budget of 40–80 evaluations, where the baselines struggle to achieve expert-level performance even with a much larger budget. This demonstrates the advantage of BaCO to deliver high performance for a small budget, even for complex search spaces.

BaCO delivers on average 2.87×–3.87× faster than the state-of-the-art autotuning frameworks. Fig. 6 highlights the quicker performance evolution of BaCO for representative benchmarks, and a more detailed breakdown is presented in Table 9 in the appendix. These results show that BaCO delivers performance much quicker than the baselines.

RQ2) Does BaCO generalize across compiler frameworks and benchmarks? Our evaluation across three diverse real-world compiler frameworks shows consistently that BaCO significantly outperforms the baselines. In fact, BaCO is the *only* framework that outperforms the expert configuration on all benchmarks across compiler frameworks, as shown in Fig. 6, with a more detailed breakdown in Table 5 in the appendix. This observation suggests that the techniques discussed in the paper generalize well across compiler frameworks and benchmarks.

RQ3) What is the performance benefit of customizing Bayesian optimization for compiler autotuning? Our system demonstrates advantages over prior work BO-based autotuning frameworks not specifically customized for compiler domains (Ytopt in Fig. 8). As explained in Sec. 3 and Sec. 4, these improvements validate our design choices and suggests that there are significant performance benefits to be gained by customizing the BO framework for this particular domain.

RQ4) What are the findings of autotuning our distinct real-world compilers using BaCO? This question attempts to provides insight on why *BaCO* outperforms baselines for our real-world compiler benchmarks. We identify three main areas: exploration of new configurations, testing schedules that did not previously exist in prior work, and better handling of both known and unknown constraints for complex real-world applications. Evaluations for this question comes from Fig. 5, Fig. 7, and Fig. 11 in the Appendix.

5.1 Baseline Methods

To contextualize the performance of BaCO, we evaluate it alongside two state-of-the-art autotuning frameworks and two random sampling approaches.

ATF with OpenTuner The Auto-Tuning Framework (ATF) [46] extends the popular OpenTuner [2] to handle known constraints. We chose OpenTuner as a baseline since it is one of the leading frameworks for autotuning.

Ytopt Ytopt [63] is an autotuning framework using BO and is part of the PROTEAS-TUNE project [1]. It supports both Random

Forests and Gaussian processes. We run it here with Random Forests as the GP implementation does not support constraints. When infeasible solutions are found due to hidden constraints, they are added to the data set with a high objective value. We compare against Ytopt since it is one of the only frameworks that supports either constraints or GP. When addressing RQ3, we run Ytopt with GPs.

Uniform and CoT random sampling These are uniform random sampling methods. The CoT random sampling is a method that randomly samples at uniform directly from the CoT. This baseline allows us to study the impact of the bias introduced by the known constraints, as explained in Sec. 4.2.

Default and expert configurations For reference, we show the performance of two baseline configurations: The *default* configuration, and an *expert* configuration, carefully handcrafted by domain experts in the respective programming languages. It is unlikely that a developer would exceed the expert performance baseline, which makes it a suitable data point for our empirical analysis. The HPVM2FPGA benchmarks are automatically generated by the autoscheduler and do not provide any expert configurations, in which case we only report the default. The expert configurations are taken from prior publications: TACO [49], RISE & ELEVATE [19, 29, 54, 57], and HPVM2FPGA [13]. The original authors used manual or semi-automated methods to determine well-performing configurations based on their experience, hardware characteristics, or data properties. The authors had the incentive to produce the best configurations but, presumably due to time constraints, they may have occasionally missed better-performing configurations.

Benchmark	Dim	Params	Constr.	Space size	Feasible	Full Budget
TACO						
SpMV	7	O/C/P		1.5×10^7	3.0×10^6	70
SpMM	6	O/C/P	K	5.2×10^{11}	4.7×10^4	60
SDDMM	6	O/C/P	K	5.2×10^{11}	7.8×10^4	60
TTV	7	O/C/P	K/H	1.5×10^7	6.0×10^6	70
MTTKRP	6	O/C/P	K	1.5×10^6	6.8×10^5	60
RISE & ELEVATE						
MM_CPU	5	O/P	K/H	1.0×10^7	2.9×10^4	100
MM_GPU	10	O	K/H	1.1×10^{11}	1.5×10^8	120
Asum_GPU	5	O	K	1.2×10^6	6.3×10^4	60
Scal_GPU	7	O	K/H	3.9×10^7	4.2×10^6	60
K-means_GPU	4	O	K/H	1.4×10^4	3.6×10^3	60
Harris_GPU	7	O	K	7.7×10^9	1.0×10^7	100
Stencil_GPU	4	O	K	1.4×10^4	3.6×10^3	60
HPVM2FPGA						
BFS	4	I/C	H	256	256	20
Audio	15	I/C	H	8.4×10^5	8.4×10^5	60
PreEuler	7	I/C	H	1.5×10^4	1.5×10^4	60

Table 3: We evaluate *BaCO* on 15 important kernels from domains like machine learning, statistics, and signal processing. The benchmarks expose search spaces with varying number of parameters (*Dim*). They cover all parameter types considered (*Params*): real (R), integer (I), ordinal (O), categorical (C), and permutation (P). *Constr.* describes the type of constraints used by the benchmark: known (K) and hidden (H) constraints. *Space size* describes the number of possible configurations in the dense search space with *Feasible* denoting all valid configurations with respect to the known constraints. *Full Budget* is the total number of evaluations we allow for autotuning the kernel.

5.2 Benchmarks

BaCO is evaluated over 15 kernels from linear algebra, machine learning, image processing, statistics, and signal processing. We integrate BaCO in the three real-world frameworks presented in Sec. 2. The benchmarks have been chosen based on prior work by the authors of the three frameworks. Furthermore, most of these benchmarks have an expert optimized code which allows for a fair comparison. The search space size ranges from tens of thousands to billions of configurations, as described in Table 3, which is beyond the scope of exhaustive search.

To define the evaluation budget for each benchmark, we first establish a full budget for each benchmark, as shown in the last column of Table 3. The full budget is defined using a rule of thumb of around 5 to 6 minutes. This max compilation time is commonly adopted in large companies, such as Google. We then define *tiny* and *small* budgets as 1/3 and 2/3 of the *full* budget.

TACO benchmarks We benchmark 5 tensor algebra expressions, commonly used in machine learning and tensor factorization [16]. Namely, they comprise of sparse matrix-vector multiply (SpMV) $a_i = \sum_k B_{ij} c_k$, sparse matrix multiply (SpMM) $A_{ij} = \sum_k B_{ik} C_{kj}$, sampled dense-dense matrix multiply (SDDMM) $A_{ij} = \sum_k B_{ij} C_{ik} D_{jk}$, tensor times vector (TTV) $A_{ij} = B_{ijk} c_k$, and fourth-order matricized tensor times Khatri-Rao product (MTTKRP) $A_{ij} = B_{iklm} * C_{kj} * D_{lj} * E_{mj}$. Each tensor expression is given a scheduling template that exposes tiling parameters (split and unrolling factors) and permutation parameters (loop reorderings). BaCO searches for the set of parameters, and therefore the schedule, that yields the best performance. The characteristics of these parameters and the search space is described in Table 3. We use tensors from a wide variety of real-world applications ranging from power networks and circuits to fluid dynamics and social networks. We run matrix expressions on a subset of SuiteSparse matrices [10, 31] and synthetic uniform random tensors, and we run higher-order expressions on the Facebook Activities tensor [59], a subset of the FROSTT tensor collection [51], and synthetic tensors as well (see Table 4). The selected tensors vary widely across tensor properties including number of nonzeroes, dense dimension size, and irregular nonzero pattern.

The TACO benchmarks were run on nodes with two Intel Xeon Gold 6130 processors locked at 2100Ghz, using all 32 cores and 96GB of RAM.

RISE & ELEVATE benchmarks We use six benchmarks covering multiple domains, optimizations, and hardware devices. This results in benchmarks requiring various autotuning features, as described in Table 3.

The CPU Matrix Multiplication (MM_CPU) benchmark from [19] is run on a CPU and applies tiling, vectorization, and loop-permutation optimizations. The remaining benchmarks are run on a GPU and apply GPU-specific optimizations, including the OpenCL-specific work-group configuration, memory hierarchies, and coalescing. The MM_GPU and K-means_GPU dense linear algebra benchmarks are inspired by implementations used in [56]. The linear algebra algorithms Asum_GPU and Scal_GPU are from [54],

Matrix	Dimension	Nonzeroes	Dataset
ACTIVSg10K	20,000 × 20,000	135,888	SS
email-Enron	36,692 × 36,692	367,662	SS
Goodwin_040	17,922 × 17,922	561,677	SS
scircuit	170,998 × 170,998	958,936	SS
filter3D	106,437 × 106,437	2,707,179	SS
laminar_duct3D	67,173 × 67,173	3,788,857	SS
cage12	130,228 × 130,228	2,032,536	SS
smt	25,710 × 25,710	3,749,582	SS
random2	10,000 × 10,000	5,000,000	Rand
random1	1000 × 500 × 100	5,000,000	Rand
facebook	1,504 × 42,390 × 39,986	737,934	FB
uber	183 × 24 × 1,140 × 1,717	3,309,490	FT
nips	2,482 × 2,482 × 14,036 × 17	3,101,609	FT
chicago	6,186 × 24 × 77 × 32	5,330,673	FT
uber3	183 × 1,140 × 1,717	1,117,629	FT*

Table 4: Tensors used in our TACO benchmarks from the SuiteSparse matrix collection (SS) [10, 31], Facebook Activities graph (FB) [59], FROSTT (FT) collection [51] or synthetically generated (Rand).
***We modify a FROSTT 4-tensor to a 3-tensor by dropping one dimension since the next largest FROSTT tensor has more than 75× the number of nonzeros.**

the stencil from [57]. The remaining image processing algorithm Harris_GPU is a corner detector described in [29].

The RISE & ELEVATE evaluation was executed on 8 cores of an Intel Xeon E5-2650 v3 @2.30Ghz processor accompanied by 32 GB of RAM. For the GPU benchmarks, we used a NVIDIA K80 GPU.

HPVM2FGA benchmarks We use the benchmarks presented in [13]: (1) Breadth First Search (BFS), and (2) the computational fluid dynamics algorithm of euler with pre-computed fluxes (PreEuler), are taken from the Rodinia Benchmark suite [6], and (3) 3D Spacial Audio Encoder (Audio) from the ILLIXR testbed [24]. The benchmarks represent diverse workloads from different domains with varying parameter space sizes, ranging from 4 parameters for BFS, to 15 for Audio.

We ran these benchmarks through HPVM2FPGA's optimizer, reporting the estimated execution time targeting an Intel Arria 10 GX FPGA in our evaluation results.

5.3 Results

RQ1) Does *BaCO* achieve high performance with a limited autotuning budget? Fig. 5 shows the average performance of BaCO and the baselines for three different levels of autotuning budget for all our benchmarks. The tiny budget is only 20-40 evaluations.[3] BaCO clearly outperforms the baselines, delivering on average better performance than all baselines for 19 out the 24 benchmarks. For TACO, the tiny budget is even sufficient to deliver expert-level performance. With the small budget, BaCO delivers expert performance for all three compiler frameworks. The baselines struggle to deliver good performance even with the full budget, particularly for the challenging spaces in the RISE benchmarks. Tables 6, 7, and 8 in the appendix show the detailed performance results for each individual benchmark and autotuning framework.

BaCO also achieves performance faster, i.e., it reaches the final performance of the other baselines using fewer configuration evaluations. Fig. 6 shows the performance evolution for three selected

[3]Besides the BFS benchmark, for which it is only 6 evaluations

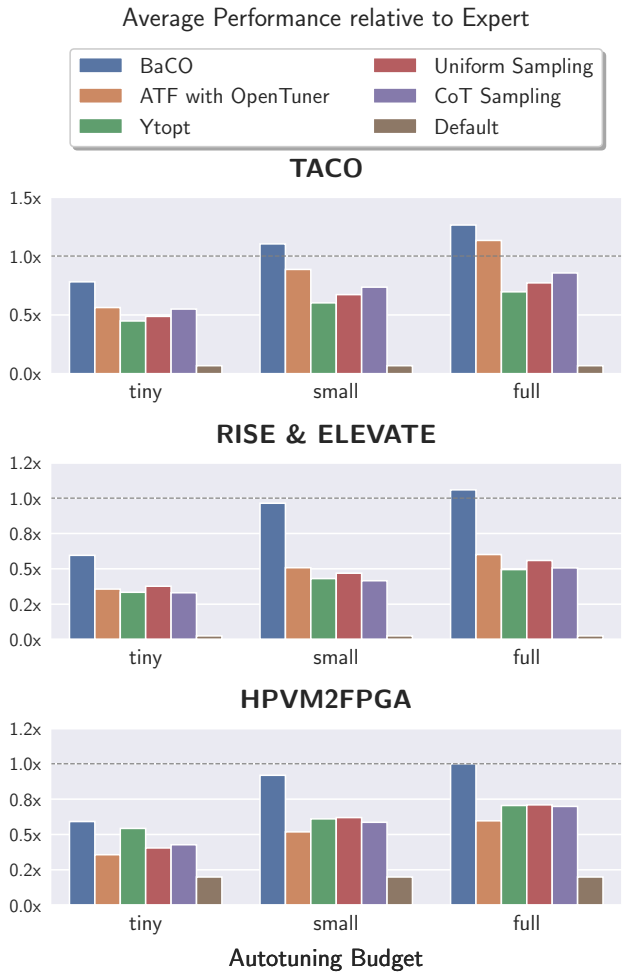

Figure 5: Average performance relative to expert with a tiny budget (1/3 of the full budget), a small budget (2/3 of the full budget) and the full budget. BaCO delivers the highest performance for each budget and achieves expert-level performance consistently with the small budget.

benchmarks, and that BaCO delivers performance with 2.9×–5× fewer evaluations than ATF and Ytopt. On average, BaCO finds the best ATF and Ytopt configurations 2.87× and 3.82× faster, respectively. Our experiments confirm that these results generalize well across our benchmarks, however, due to space constraints these additional results are presented in Table 6 in the Appendix.

RQ2) Does BaCO generalize across compiler frameworks? To see how the performance generalizes, we show in Fig. 7 (and Fig. 11 in the Appendix) how the mean of the best-found solution by each framework improves over time for each individual benchmark. In the figures, the average performance is plotted for each method and each benchmark. The goal is to achieve a lower value, which identifies a better-performing configuration, and to find low values as far to the left as possible, which means using a low tuning budget. The performance of the default configuration and expert

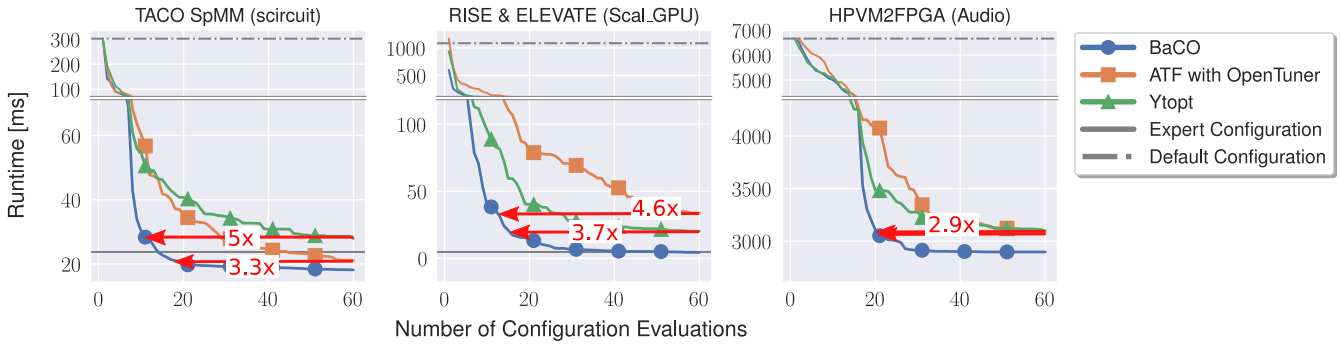

Figure 6: Evolution of average best runtime for one kernel from each framework. The figure is split vertically into two different scales. BaCO reaches the final performance of the state-of-the-art methods using as little as 30% of the function evaluations of the other methods.

configuration is presented for reference when available. We further denote when each method reaches expert-level performance with a star, such that a shorter distance between the y-axis and the star is better. As finding improvement over the default configuration initially is easy, we split the plots into two regions with different scales. This helps focus on the interesting part closer to the expert-level performance. *BaCO* reliably yields high-level performance and overall provides the best schedule in 22 out of the 24 benchmarks. It is further frequently the only method to reach expert-level performance within the given budget.

RQ3) What is the performance benefit of customizing Bayesian optimization for compiler autotuning? We study the *BaCO* design by running three matrices for SpMM with default settings and with a number of major features turned off. We use the SpMM benchmark as it is reasonably well-behaved and only has few constraints. The average speedup over expert is shown in Fig. 8. We denote the restricted version by *BaCO*--. In more detail, *BaCO*-- is *BaCO* without variable transformations, model priors, and local search for the acquisition function. It further uses the naive distance for permutation variables that ignores their underlying structure and it does not use *BaCO*'s more advanced fitting of the GP hyperparameters. We see that by doing those changes, *BaCO* takes about a 20% performance loss.

Next, we compare it with the GP implementation of Ytopt. Ytopt uses none of the above mentioned features, but additionally has a less advanced GP and BO toolkit. Ytopt only supports constraints for RF so this Ytopt configuration does not support constraints. However, for this benchmark we have manually pruned the search space for Ytopt, so that the only remaining constraint is a single less-than relationship between two variables. BO with GPs requires a lot of care to be efficient, which we see from the difference between Ytopt (GP) and *BaCO*--.

Lastly, we show the difference between a well implemented GP and RF as surrogate model. Especially for smaller budgets, the GP model shows much stronger performance. This is relevant as there is currently a paradigm shift towards using GPs in discrete settings.

Ablation analysis To further understand the impact of the different design choices in BaCO, we perform an ablation analysis in Fig. 9. The impact of the permutation kernel, variable transformations and model priors are studied in an ablation analysis. First, BaCO in default settings is presented, which is using Spearman's rank correlation for permutation variables. Then we study the impact of changing the permutation metric to Kendall distance, Hamming distance, as well as the naive approach of treating permutations as categorical variables (Sec. 4.1). The Spearman metric yields the best performance, especially in early iterations.

Secondly, we study the impact of removing the logarithmic transformations of variables and output (Sec. 4.2), and the model priors (Sec. 3.2). Removing the log transforms significantly deteriorates the performance at all evaluation counts. The lengthscale priors, however, have a larger impact early on, where they work to stabilize the procedure, and become less important when more data has been observed with which to fit the model.

Overall, we see that the changes have a much larger impact early on, but except for transformations, ignoring any of the individual features fails to prevent good performance after more iterations. It is noteworthy that no individual design choice has a major impact, but together they make a large difference.

Hidden constraints Next, we study the impact of the predicting feasibility with respect to hidden constraints on two benchmarks from the RISE/ELEVATE suite. In Figure 10, we show the average improvement over expert after 20, 40, 60 iterations with and without the feasibility predictor. Additionally, we show the impact of the minimum feasibility limit presented in Sec. 4.2. It shows that the hidden constraints predictor has a significant positive impact, particularly after more iterations where it has had more samples to train on. But it also indicates that the introduced minimum feasibility limit (Sec. 4.2) is important to stabilize the interaction between the feasibility predictor and the surrogate model.

Chain-of-Trees Even after manual sparsity-reducing transformations, some of the search spaces remain highly sparse. When this is the case, CoTs greatly increases the efficiency of sampling from, and searching, the parameter domain. On the MM_GPU search space for example, over ten runs, using the CoTs reduced the time spent on evaluating constraints in the local search by a factor 6× and the random sampling by a factor of 80×. Overall, this resulted in that the time spent by the internal working of *BaCO* was reduced

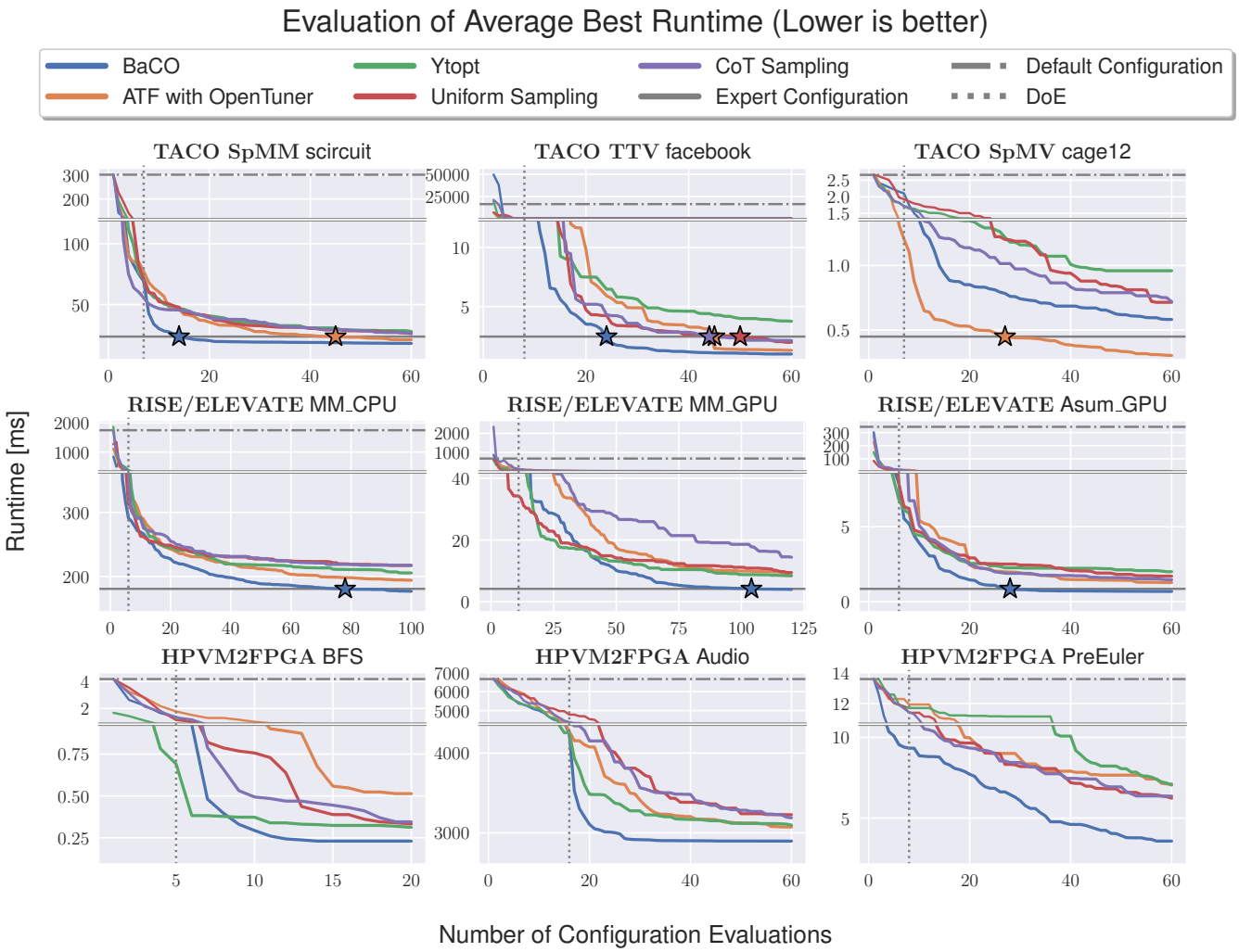

Figure 7: Evolution of average best runtime among evaluated configurations for selected benchmarks from Table 3. Fig. 11 in the Appendix contains from Table 3 not shown here. The figure is split vertically into two different scales, and we mark the iteration where each method beats the expert configuration with a star.

by 70%. For even more sparse search spaces, operating directly on the search space quickly becomes untenable.

RQ4) What are the findings of autotuning our distinct real-world compilers using *BaCO*?

Configuration insight BaCO underperforms baselines in only 1 of our 24 (4%) benchmarks across the 15 kernels. Opentuner is able to beat out BaCO (as shown in Fig. 7) when running SpMV on cage12 (see Table 4). The SpMV benchmark is interesting as it has a good default setting, but ill-designed schedules can increase the run time by several orders of magnitude. After inspection of the configurations, it shows that ATF picks configurations similar to its previous configurations. This behavior in ATF exploits sampling around prior good configurations each evaluation, whereas BaCO's algorithm is more explorative in finding completely new configurations. Exploiting configurations works for simple kernels, like SpMV, but fails for real-world kernels with increased complexity and runtime variance. Exploitation sampling is likely to get stuck in a local minima, which is more likely to be globally bad for complex problems (as is the case with TTV on the random1 in Fig. 11 in the Appendix). We do not augment BaCO in any specific manner to explore configurations, but the global nature of the BO paradigm emphasizes exploration over methods with local-exploration elements such as OpenTuner.

Performance over expert BaCO is able to achieve better than expert performance in some cases (see Fig. 5). Even experts in the domain, with insight of the underlying hardware architecture, may miss out on the optimal configuration and the best performance simply due to the amount of user-time needed to explore the vast search space of possible configurations. BaCO allows users to automate that search while potentially offering better performance than the expert could find. For example, BaCO is able to find over a 1.1× speedup on average for TACO (see Fig. 5) since experts in [49]

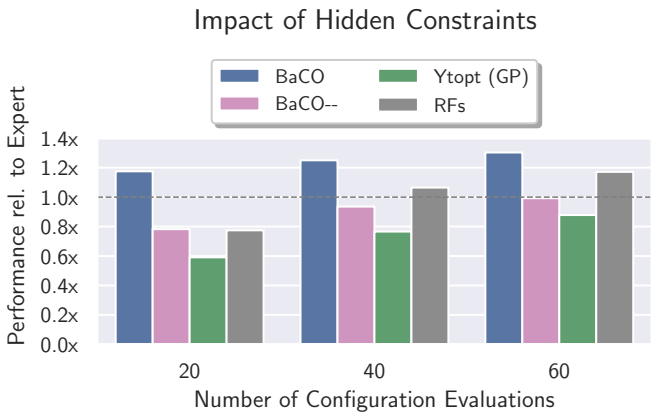

Figure 8: Geometric mean of the performance relative to the expert configuration for the TACO SpMM kernel applied to the *filter3D, email-Enron* and *amazon0312* matrices after 20, 40 and 60 evaluations.

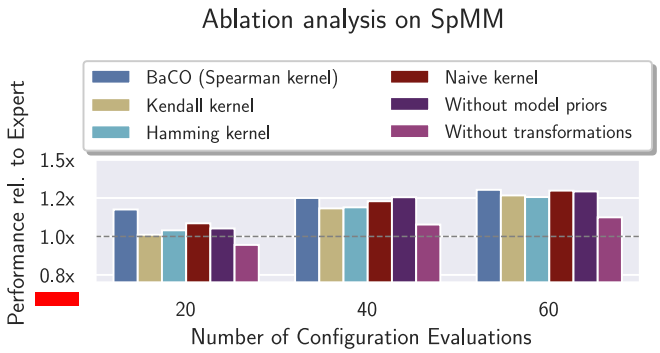

Figure 9: Geometric mean of the performance relative to the expert configuration for the SpMM kernel applied to the *filter3D, email-Enron* and *amazon0312* matrices after 20, 40 and 60 evaluations. (Note the cut axis).

only considered the default loop ordering (permutation) for all expressions. In addition, many of the configurations the autotuner discovers are hard to find by hand due to the concordant traversal of a compressed tensor data structure. Therefore, it is difficult for an expert to search the space of loop orderings and know which of them are infeasible.

Constraints in real-world applications Over half (8/15) of our benchmarks, and notably all of the HPVM2FPGA benchmarks, use hidden constraints. Additionally, all but one benchmark uses known constraints, significantly reducing the feasible search space as shown by the Feasible column in Table 3. Predictor modeling of hidden constraints has a significant impact on performance (as discussed in RQ3), and this impact is apparent in our real-world compiler benchmarks.

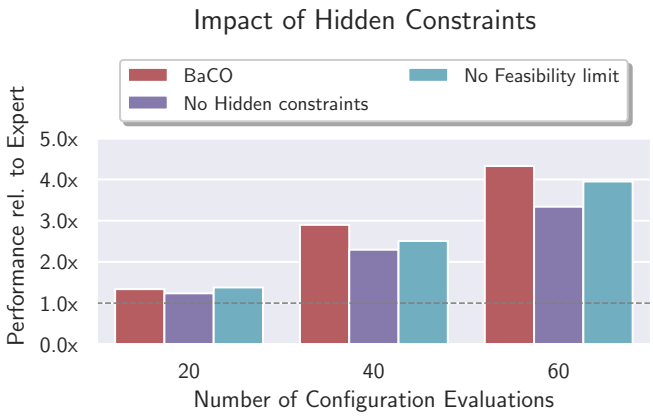

Figure 10: Geometric mean of the performance relative to the expert configuration for the MM_GPU and Scal_GPU kernels after 20, 40 and 60 evaluations.

6 RELATED WORK

Bayesian optimization for autotuning Several Bayesian optimization frameworks have been presented for autotuning [38, 41, 61, 63]. One of the earlier frameworks was introduced by Nelson *et al.* [42], who present *SURF* that uses Random Forests models to optimize tensor constraction operations on GPUs. This work is later extended to become *Ytopt* by Wu *et al.* [63]. The authors use the skopt Bayesian optimization framework to optimize LLVM Clang/Polly pragma configurations on the PolyBench benchmark suite. Ytopt further allows the usage of additional surrogate models such as Gaussian Processes and Boosted Trees. Ytopt implements a method based on Bayesian optimization, which BaCO builds on. However, we show that the Bayesian optimization pipeline needs to be further customized to work well on autotuning domains, which is the scope of our work. The work by Sid-Lakhdar *et al.* [50] focuses on the meta- and multi-task learning aspect. It was extended by Liu *et al.* [38] into the *GPTune* framework. The authors use *linear coregionalization models* (LCMs), to model multiple similar problems simultaneously to increase efficiency. This was further extended by Zhu *et al.* [65] to also handle multifidelity applications. While this is out of the scope of the current work, the use of meta-learning can be used in combination to BaCO to achieve greater efficiency. Willems *et al.* [61] use Bayesian optimization to autotune GPU kernels using Gaussian Processes and known constraints on the search space. Another recent approach is *Bliss* by Roy *et al.* [48], that probabilistically chooses a combination of models and acquisition functions each new optimization iteration based on previous performance observations. *Bliss'* approach is orthogonal to BaCO and it is possible that combining the methods further efficiency can be achieved. Recently, Dorier *et al.* [11] present DeepHyper, a Bayesian optimization framework for HPC storage system autotuning, that focuses on transfer learning through the use of variational autoencoders.

Bayesian optimization for design space exploration Nardi *et al.* [30, 41] use Bayesian optimization with a RFs surrogate model to optimize FPGAs. They consider both multiobjective and hidden

constraints. Ejjeh *et al.* [13] use the same DSE framework to tune hardware-agnostic programs targeting FPGA backends. [53] design a human-centric DSE approach, where expert priors accelerate the convergence of the autotuner. While this is not the focus of our work, a simple adaptation of the BaCO acquisition function can benefit the same user priors when available.

There is substantial work in the literature about DSE techniques in HLS [5, 14, 60, 62, 64]. However, most existing work focuses on using DSE for tuning HLS, rather than using it to select compiler optimizations [5, 14, 64]. These works are not based on Bayesian optimization and we view them as complementary to our work.

The phase-ordering problem Autoscheduling tackles the task of applying a number of transformations to optimize a kernel automatically. Typically, the scheduling language parametrizes the application of those transformations by a bounded set of options which we refer to as parameters that are easier to optimize over. This parametrization approach is the one used by TACO and ELEVATE. However, a different approach is to operate directly on the space of transformations. Optimization over this unbounded tree-like space is commonly known as the *phase-ordering problem* [21, 22, 32, 34].

7 CONCLUSIONS AND FUTURE WORK

We introduce the Bayesian Compiler Optimization framework (BaCO), a plug-and-play solution to autoscheduling tasks for modern scheduling languages targeting various hardware backends. BaCO is able to reach expert-level performance 2.7×-10× faster than the state of the art autotuners. The separation of concerns between policy and mechanism allows compiler users to delegate the complex and time-consuming task of scheduling to BaCO so that they can focus on their applications instead.

While we show that BaCO can provide high-performing solutions in less than 100 seconds, this time is still too long for use in software development. The holy grail of autoscheduling is to be able to use an autotuner during the development, and ideally enable the user to run autotuning every time they compile their code. That way users can check both functional and non-functional properties on a regular basis during the various program lifecycle phases. Indeed, increasing the efficiency of the autotuner would enable a new level of autotuning-in-development-loop paradigm which is not accessible with the current state of autotuning technology.

ACKNOWLEDGEMENTS

We would like to thank Jaeyeon Won for their help with evaluation. We would like to thank Michael O'Boyle, Tobias Grosser and Jacob Odgård Tørring for their valuable feedback on a draft of this paper. Luigi Nardi and Kunle Olukotun were supported in part by affiliate members and other supporters of the Stanford DAWN project — Ant Financial, Facebook, Google, Intel, Microsoft, NEC, SAP, Teradata, and VMware. Luigi Nardi was also supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation. Luigi Nardi was partially supported by the Wallenberg Launch Pad (WALP) grant Dnr 2021.0348. The computations were also enabled by resources provided by the Swedish National Infrastructure for Computing (SNIC) at LUNARC, partially funded by the Swedish Research Council through grant agreement no. 2018-05973. Olivia Hsu was supported by an NSF GRFP Fellowship. This research was also supported in part by the National Science Foundation under Grant CCF-2143061, 1937301, 2028602, CCF-1563078, and 1563113 and the Google Research Scholar program. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the aforementioned funding agencies.

REFERENCES

[1] Proteas-tune. https://www.ornl.gov/project/proteas-tune. Accessed: 2022-10-18.
[2] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. OpenTuner: An extensible framework for program autotuning. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2014.
[3] The GPyOpt authors. GPyOpt: A bayesian optimization framework in python. http://github.com/SheffieldML/GPyOpt, 2016.
[4] Prasanna Balaprakash, Jack Dongarra, Todd Gamblin, Mary Hall, Jeffrey K Hollingsworth, Boyana Norris, and Richard Vuduc. Autotuning in high-performance computing applications. *Proceedings of the IEEE*, 106(11):2068–2083, 2018.
[5] Pedro Bruel, Alfredo Goldman, Sai Rahul Chalamalasetti, and Dejan Milojicic. Autotuning high-level synthesis for fpgas using opentuner and legup. In *International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE, 2017.
[6] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *International Symposium on Workload Characterization (IISWC)*. IEEE, 2009.
[7] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. TVM: An automated End-to-End optimizing compiler for deep learning. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.
[8] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. Format abstraction for sparse tensor algebra compilers. *Proc. ACM Program. Lang.*, 2(OOPSLA):123:1–123:30, October 2018.
[9] Jhouben Cuesta Ramirez, Rodolphe Le Riche, Olivier Roustant, Guillaume Perrin, Cedric Durantin, and Alain Gliere. A comparison of mixed-variables bayesian optimization approaches. *Advanced Modeling and Simulation in Engineering Sciences*, 9(1):1–29, 2022.
[10] Timothy A Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1–25, 2011.
[11] Matthieu Dorier, Romain Egele, Prasanna Balaprakash, Jaehoon Koo, Sandeep Madireddy, Srinivasan Ramesh, Allen D Malony, and Rob Ross. Hpc storage service autotuning using variational-autoencoder-guided asynchronous bayesian optimization. *arXiv preprint arXiv:2210.00798*, 2022.
[12] Adel Ejjeh, Aaron Councilman, Akash Kothari, Maria Kotsifakou, Leon Medvinsky, Abdul Rafae Noor, Hashim Sharif, Yifan Zhao, Sarita Adve, Sasa Misailovic, et al. HPVM: Hardware-agnostic programming for heterogeneous parallel systems. *IEEE Micro*, 42(5):108–117, 2022.
[13] Adel Ejjeh, Leon Medvinsky, Aaron Councilman, Hemang Nehra, Suraj Sharma, Vikram Adve, Luigi Nardi, Eriko Nurvitadhi, and Rob A Rutenbar. HPVM2FPGA: Enabling true hardware-agnostic fpga programming. In *IEEE International Conference on Application-specific Systems, Architectures, and Processors (ASAP)*, 2022.
[14] Lorenzo Ferretti, Andrea Cini, Georgios Zacharopoulos, Cesare Alippi, and Laura Pozzi. A graph deep learning framework for high-level synthesis design space exploration, 2021.
[15] Peter I Frazier. A tutorial on bayesian optimization. *arXiv preprint arXiv:1807.02811*, 2018.
[16] Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. Sparse gpu kernels for deep learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20. IEEE Press, 2020.
[17] Jacob R Gardner, Matt J Kusner, Zhixiang Eddie Xu, Kilian Q Weinberger, and John P Cunningham. Bayesian optimization with inequality constraints. In *International Conference on Machine Learning (ICML)*, 2014.
[18] Eduardo C Garrido-Merchán and Daniel Hernández-Lobato. Dealing with categorical and integer-valued variables in bayesian optimization with gaussian processes. *Neurocomputing*, 380:20–35, 2020.
[19] Bastian Hagedorn, Johannes Lenfers, Thomas Koehler, Xueying Qin, Sergei Gorlatch, and Michel Steuwer. Achieving high-performance the functional way: a functional pearl on expressing high-performance optimizations as rewrite strategies. *Proceedings of the ACM on Programming Languages*, 4(ICFP):1–29, 2020.
[20] Bastian Hagedorn, Larisa Stoltzfus, Michel Steuwer, Sergei Gorlatch, and Christophe Dubach. High performance stencil code generation with Lift. In

International Symposium on Code Generation and Optimization (CGO), 2018.

[21] Ameer Haj-Ali, Hasan Genc, Qijing Huang, William Moses, John Wawrzynek, Krste Asanović, and Ion Stoica. Protuner: tuning programs with monte carlo tree search. *arXiv preprint arXiv:2005.13685*, 2020.

[22] Ameer Haj-Ali, Qijing Jenny Huang, John Xiang, William Moses, Krste Asanovic, John Wawrzynek, and Ion Stoica. Autophase: Juggling hls phase orderings in random forests with deep reinforcement learning. *Proceedings of Machine Learning and Systems*, 2:70–81, 2020.

[23] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *International conference on learning and intelligent optimization (LION)*, 2011.

[24] Muhammad Huzaifa, Rishi Desai, Samuel Grayson, Xutao Jiang, Ying Jing, Jae Lee, Fang Lu, Yihan Pang, Joseph Ravichandran, Finn Sinclair, et al. ILLIXR: Enabling end-to-end extended reality research. In *2021 IEEE International Symposium on Workload Characterization (IISWC)*, pages 24–38. IEEE, 2021.

[25] Donald R Jones, Matthias Schonlau, and William J Welch. Efficient global optimization of expensive black-box functions. *Journal of Global optimization*, 13(4):455–492, 1998.

[26] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA), 2017.

[27] Aaron Klein, Stefan Falkner, Simon Bartels, Philipp Hennig, and Frank Hutter. Fast bayesian optimization of machine learning hyperparameters on large datasets. In *Artificial intelligence and statistics*, pages 528–536. PMLR, 2017.

[28] Nicolas Knudde, Joachim van der Herten, Tom Dhaene, and Ivo Couckuyt. GPflowOpt: A Bayesian Optimization Library using TensorFlow. *arXiv preprint – arXiv:1711.03845*, 2017.

[29] Thomas Koehler and Michel Steuwer. Towards a domain-extensible compiler: optimizing an image processing pipeline on mobile cpus. In *International Symposium on Code Generation and Optimization (CGO)*, 2021.

[30] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, et al. Spatial: A language and compiler for application accelerators. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 296–311, 2018.

[31] Scott P Kolodziej, Mohsen Aznaveh, Matthew Bullock, Jarrett David, Timothy A Davis, Matthew Henderson, Yifan Hu, and Read Sandstrom. The suitesparse matrix collection website interface. *Journal of Open Source Software*, 4(35):1244, 2019.

[32] Jaehoon Koo, Prasanna Balaprakash, Michael Kruse, Xingfu Wu, Paul Hovland, and Mary Hall. Customized monte carlo tree search for llvm/polly's composable loop optimization transformations. In *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, 2021.

[33] Maria Kotsifakou, Prakalp Srivastava, Matthew D Sinclair, Rakesh Komuravelli, Vikram Adve, and Sarita Adve. Hpvm: Heterogeneous parallel virtual machine. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 68–80, 2018.

[34] Michael Kruse, Hal Finkel, and Xingfu Wu. Autotuning search space for loop transformations. In *IEEE/ACM Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC) and Workshop on Hierarchical Parallelism for Exascale Computing (HiPar)*, 2020.

[35] M. Lindauer, K. Eggensperger, M. Feurer, A. Biedenkapp, J. Marben, P. Müller, and F. Hutter. Boah: A tool suite for multi-fidelity bayesian optimization & analysis of hyperparameters. *arXiv:1908.06756 [cs.LG]*.

[36] Marius Lindauer, Katharina Eggensperger, Matthias Feurer, André Biedenkapp, Difan Deng, Carolin Benjamins, Tim Ruhkopf, René Sass, and Frank Hutter. SMAC3: A versatile bayesian optimization package for hyperparameter optimization. *Journal of Machine Learning Research*, 23(54):1–9, 2022.

[37] Dong C Liu and Jorge Nocedal. On the limited memory BFGS method for large scale optimization. *Mathematical Programming*, 45(1):503–528, 1989.

[38] Yang Liu, Wissam M Sid-Lakhdar, Osni Marques, Xinran Zhu, Chang Meng, James W Demmel, and Xiaoye S Li. GPTune: multitask learning for autotuning exascale applications. In *Principles and Practice of Parallel Programming (PPoPP)*, 2021.

[39] Maria Lomeli, Mark Rowland, Arthur Gretton, and Zoubin Ghahramani. Antithetic and monte carlo kernel estimators for partial rankings. *Statistics and Computing*, 29(5):1127–1147, 2019.

[40] Kevin P Murphy. *Machine learning: a probabilistic perspective*. MIT press, 2012.

[41] Luigi Nardi, David Koeplinger, and Kunle Olukotun. Practical design space exploration. In *Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2019.

[42] Thomas Nelson, Axel Rivera, Prasanna Balaprakash, Mary Hall, Paul D Hovland, Elizabeth Jessup, and Boyana Norris. Generating efficient tensor contractions for gpus. In *International Conference on Parallel Processing (ICPP)*, 2015.

[43] Filip Petrovič, David Střelák, Jana Hozzová, Jaroslav Ol'ha, Richard Trembecký, Siegfried Benkner, and Jiří Filipovič. A benchmark set of highly-efficient cuda and opencl kernels and its dynamic autotuning with kernel tuning toolkit. *Future Generation Computer Systems*, 108:161–177, 2020.

[44] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM Sigplan Notices*, 48(6):519–530, 2013.

[45] Ari Rasch, Michael Haidl, and Sergei Gorlatch. 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)*, pages 64–71. IEEE, 2017.

[46] Ari Rasch, Richard Schulze, Michel Steuwer, and Sergei Gorlatch. Efficient auto-tuning of parallel programs with interdependent tuning parameters via auto-tuning framework (ATF). *ACM Transactions on Architecture and Code Optimization*, 18(1):1:1–1:26, 2021.

[47] Carl Edward Rasmussen and Christopher K. I. Williams. *Gaussian processes for machine learning*. MIT press, 2006.

[48] Rohan Basu Roy, Tirthak Patel, Vijay Gadepally, and Devesh Tiwari. Bliss: auto-tuning complex applications using a pool of diverse lightweight learning models. In *Programming Language Design and Implementation (PLDI)*, 2021.

[49] Ryan Senanayake, Changwan Hong, Ziheng Wang, Amalee Wilson, Stephen Chou, Shoaib Kamil, Saman Amarasinghe, and Fredrik Kjolstad. A sparse iteration space transformation framework for sparse tensor algebra. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA), 2020.

[50] Wissam M Sid-Lakhdar, Mohsen Mahmoudi Aznaveh, Xiaoye S Li, and James W Demmel. Multitask and transfer learning for autotuning exascale applications. *arXiv preprint arXiv:1908.05792*, 2019.

[51] Shaden Smith, Jee W. Choi, Jiajia Li, Richard Vuduc, Jongsoo Park, Xing Liu, and George Karypis. FROSTT: The formidable repository of open sparse tensors and tools, 2017.

[52] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. 2012.

[53] Artur Souza, Luigi Nardi, Leonardo B Oliveira, Kunle Olukotun, Marius Lindauer, and Frank Hutter. Bayesian optimization with a prior for the optimum. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 2021.

[54] Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach. Generating performance portable code using rewrite rules: from high-level functional expressions to high-performance opencl code. *ACM SIGPLAN Notices*, 50(9):205–217, 2015.

[55] Michel Steuwer, Thomas Koehler, Bastian Köpcke, and Federico Pizzuti. RISE & shine: Language-oriented compiler design. *CoRR*, abs/2201.03611, 2022.

[56] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. Lift: a functional data-parallel ir for high-performance gpu code generation. In *Code Generation and Optimization (CGO)*, 2017.

[57] Larisa Stoltzfus, Bastian Hagedorn, Michel Steuwer, Sergei Gorlatch, and Christophe Dubach. Tiling optimizations for stencil computations using rewrite rules in lift. *ACM Trans. Archit. Code Optim.*, 16(4), dec 2019.

[58] Hakki Mert Torun, Madhavan Swaminathan, Anto Kavungal Davis, and Mohamed Lamine Faycal Bellaredj. A global bayesian optimization algorithm and its application to integrated system design. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 26(4):792–802, 2018.

[59] Bimal Viswanath, Alan Mislove, Meeyoung Cha, and Krishna P Gummadi. On the evolution of user interaction in facebook. In *Proceedings of the 2nd ACM workshop on Online social networks*, pages 37–42, 2009.

[60] Jie Wang, Licheng Guo, and Jason Cong. AutoSA: a polyhedral compiler for high-performance systolic arrays on FPGA. In *SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2021.

[61] Floris-Jan Willemsen, Rob van Nieuwpoort, and Ben van Werkhoven. Bayesian optimization for auto-tuning gpu kernels. In *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, 2021.

[62] Nan Wu, Yuan Xie, and Cong Hao. Ironman: Gnn-assisted design space exploration in high-level synthesis via reinforcement learning. In *Great Lakes Symposium on VLSI*, 2021.

[63] Xingfu Wu, Michael Kruse, Prasanna Balaprakash, Hal Finkel, Paul Hovland, Valerie Taylor, and Mary Hall. Autotuning PolyBench benchmarks with LLVM Clang/Polly loop optimization pragmas using bayesian optimization (extended version). *arXiv preprint arXiv:2104.13242*, 2021.

[64] Guanwen Zhong, Alok Prakash, Yun Liang, Tulika Mitra, and Smail Niar. Lin-Analyzer: A high-level performance analysis tool for fpga-based accelerators. In *ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2016.

[65] Xinran Zhu, Yang Liu, Pieter Ghysels, David Bindel, and Xiaoye S Li. GPTuneBand: Multi-task and multi-fidelity autotuning for large-scale high performance computing applications. In *Parallel Processing for Scientific Computing*, 2022.

A ADDITIONAL RESULTS

In this Appendix, we present the results shown in Sec. 5.3 in greater detail.

Fig. 11 shows the remaining benchmarks not in Fig. 7. The takeaway is similar, that BaCO finds the expert-level-performing configurations much faster than the other benchmarks, and that it consistently finds better configurations than the other benchmarks after only a few iterations after the learning phase starts. However, the additional benchmarks indicates that it generalizes well.

Table 5 shows for each autotuning framework and benchmark how many individual autotuning runs with a full budget managed to reach the expert performance. The results show that BaCO reaches the expert-level performance in 575 out of 750 runs (76%).

Tables 6, 7, and 8 show the relative performance achieved compared to an expert with the tiny, small, and full budgets. Table 9 presents the factors showing how much faster BaCO reaches the best performance of the other methods. On average, BaCO reaches ATF's performance 2.87× faster and Ytopt's performance 3.87× faster.

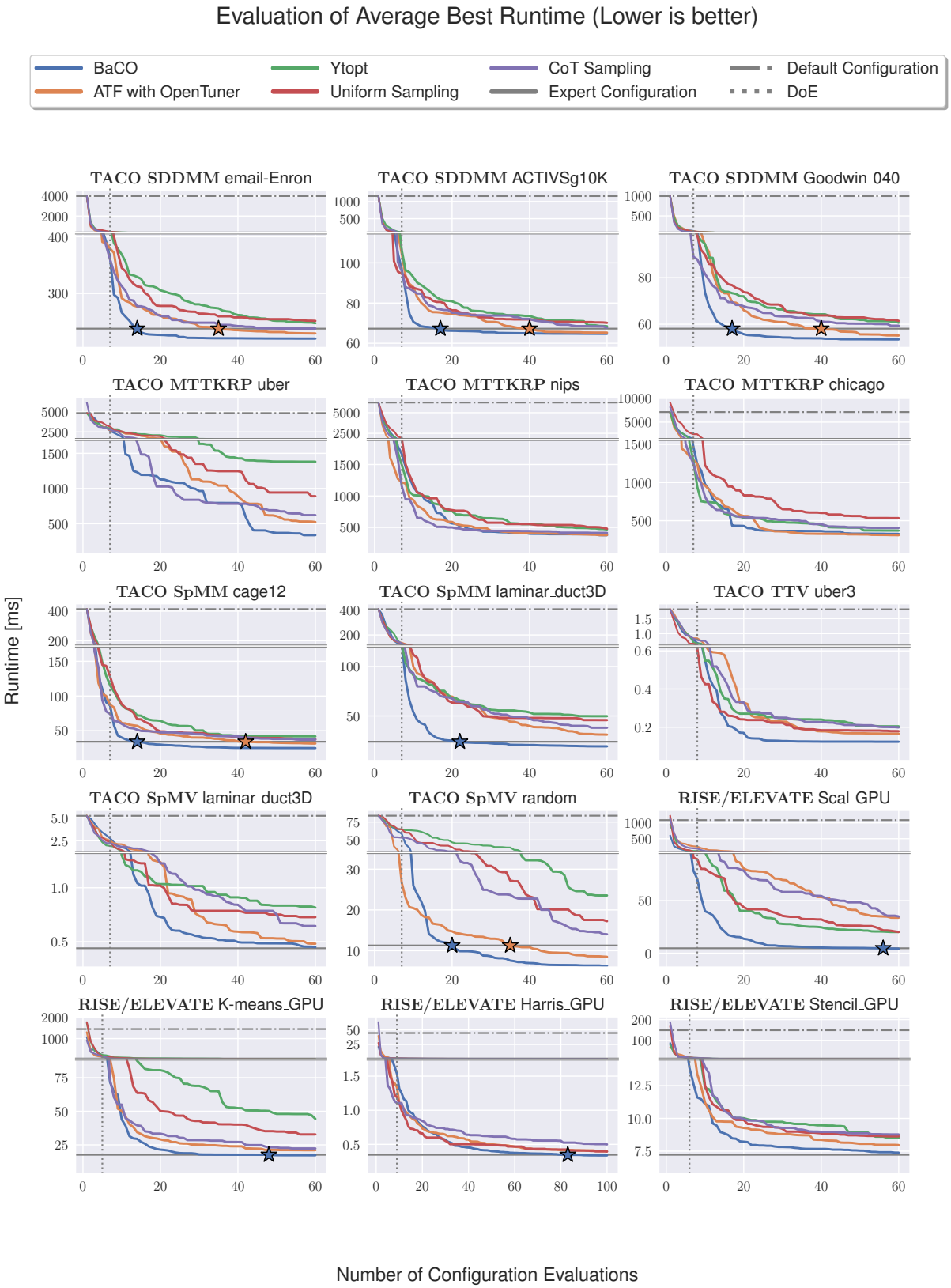

Figure 11: Evolution of average best runtime among evaluated configurations for all benchmarks in Table 3 that are not shown in Fig. 7. The figure is split vertically into two different scales and the iteration where each method beats the expert configuration is marked with a star.

Framework	Benchmark	BO	ATF	Ytopt	Uniform	CoT
TACO (SpMM)	scircuit	30	24	16	14	12
TACO (SpMM)	cage12	30	22	8	12	13
TACO (SpMM)	laminarduct3D	30	18	6	6	9
TACO (SDDMM)	email-Enron	30	25	17	10	18
TACO (SDDMM)	ACTIVSg10K	30	28	19	11	14
TACO (SDDMM)	Goodwin040	30	24	18	10	12
TACO (MTTKRP)	uber	24	21	4	3	6
TACO (MTTKRP)	nips	24	20	9	7	15
TACO (MTTKRP)	chicago	21	22	14	0	0
TACO (TTV)	facebook	30	27	16	22	20
TACO (TTV)	uber3	16	14	3	3	3
TACO (TTV)	random	18	17	16	17	17
TACO (SpMV)	laminarduct3D	13	14	1	1	3
TACO (SpMV)	cage12	13	27	8	9	8
TACO (SpMV)	filter3D	9	27	3	4	2
TACO		348	330	158	129	152
RISE & ELEVATE	MM-CPU	22	15	6	2	1
RISE & ELEVATE	MM-GPU	19	10	1	1	0
RISE & ELEVATE	Asum-GPU	29	9	5	3	3
RISE & ELEVATE	Scal-GPU	21	1	0	2	1
RISE & ELEVATE	K-means-GPU	25	15	3	5	6
RISE & ELEVATE	Stencil-GPU	18	4	8	0	0
RISE & ELEVATE		149	68	23	23	13
HPVM2FPGA	BFS	29	0	4	2	7
HPVM2FPGA	Audio	23	4	1	0	0
HPVM2FPGA	PreEuler	26	6	4	6	4
HPVM2FPGA		78	10	9	8	11

Table 5: Out of 30 autotuning runs with the full budget, how many reached expert-level performance.

Framework	Benchmark	BaCO	ATF	Ytopt	Uniform	CoT
TACO (SpMM)	scircuit	1.16	0.67	0.58	0.59	0.61
TACO (SpMM)	cage12	1.14	0.74	0.53	0.70	0.72
TACO (SpMM)	laminarduct3D	0.96	0.34	0.35	0.38	0.35
TACO (SDDMM)	email-Enron	1.05	0.91	0.78	0.86	0.91
TACO (SDDMM)	ACTIVSg10K	1.02	0.90	0.83	0.88	0.89
TACO (SDDMM)	Goodwin040	1.04	0.88	0.81	0.79	0.86
TACO (MTTKRP)	uber	0.30	0.20	0.16	0.17	0.33
TACO (MTTKRP)	nips	0.67	0.64	0.50	0.49	0.75
TACO (MTTKRP)	chicago	0.76	0.58	0.61	0.39	0.60
TACO (TTV)	facebook	0.70	0.25	0.33	0.52	0.47
TACO (TTV)	uber3	0.73	0.38	0.46	0.51	0.38
TACO (TTV)	random	1.30	0.86	0.35	0.92	1.26
TACO (SpMV)	laminarduct3D	0.60	0.35	0.42	0.43	0.36
TACO (SpMV)	cage12	0.53	0.85	0.33	0.30	0.39
TACO		0.83	0.63	0.49	0.54	0.61
RISE & ELEVATE	MM-CPU	0.89	0.81	0.79	0.78	0.78
RISE & ELEVATE	MM-GPU	0.28	0.17	0.28	0.25	0.14
RISE & ELEVATE	Asum-GPU	0.59	0.35	0.34	0.29	0.35
RISE & ELEVATE	Scal-GPU	0.34	0.06	0.12	0.11	0.06
RISE & ELEVATE	K-means-GPU	0.82	0.61	0.22	0.35	0.53
RISE & ELEVATE	Stencil-GPU	0.88	0.78	0.72	0.73	0.73
RISE & ELEVATE		0.65	0.48	0.41	0.46	0.44
HPVM2FPGA	BFS	0.48	0.18	0.60	0.28	0.29
HPVM2FPGA	Audio	0.93	0.71	0.83	0.64	0.70
HPVM2FPGA	PreEuler	0.46	0.36	0.32	0.37	0.38
HPVM2FPGA		0.62	0.42	0.58	0.43	0.46
All		0.76	0.56	0.49	0.50	0.55

Table 6: Relative performance compared to expert with the tiny budget (1/3 of the full budget). Values larger than 1 indicate a performance advantage over the expert. Values below 1 indicate a performance disadvantage compared with the expert.

Framework	Benchmark	BaCO	ATF	Ytopt	Uniform	CoT
TACO (SpMM)	scircuit	1.25	0.96	0.77	0.78	0.77
TACO (SpMM)	cage12	1.35	0.99	0.80	0.84	0.82
TACO (SpMM)	laminarduct3D	1.16	0.60	0.45	0.50	0.51
TACO (SDDMM)	email-Enron	1.08	1.01	0.91	0.92	0.98
TACO (SDDMM)	ACTIVSg10K	1.04	1.01	0.91	0.94	0.93
TACO (SDDMM)	Goodwin040	1.08	1.00	0.90	0.91	0.95
TACO (MTTKRP)	uber	0.42	0.38	0.24	0.27	0.43
TACO (MTTKRP)	nips	0.94	0.92	0.68	0.68	0.85
TACO (MTTKRP)	chicago	0.90	1.00	0.74	0.54	0.72
TACO (TTV)	facebook	2.13	0.75	0.53	0.93	0.87
TACO (TTV)	uber3	1.00	0.69	0.52	0.67	0.55
TACO (TTV)	random	6.29	1.42	0.84	1.83	2.40
TACO (SpMV)	laminarduct3D	0.88	0.74	0.48	0.56	0.52
TACO (SpMV)	cage12	0.65	1.17	0.44	0.48	0.54
TACO (SpMV)	filter3D	0.66	1.28	0.42	0.34	0.43
TACO		1.39	0.93	0.64	0.75	0.82
RISE & ELEVATE	MM-CPU	0.98	0.90	0.85	0.81	0.82
RISE & ELEVATE	MM-GPU	0.84	0.38	0.40	0.36	0.19
RISE & ELEVATE	Asum-GPU	1.20	0.58	0.38	0.41	0.49
RISE & ELEVATE	Scal-GPU	0.88	0.09	0.19	0.15	0.09
RISE & ELEVATE	K-means-GPU	1.00	0.74	0.34	0.44	0.65
RISE & ELEVATE	Stencil-GPU	0.94	0.86	0.76	0.81	0.80
RISE & ELEVATE	Harris-GPU	0.95	0.79	0.80	0.61	
RISE & ELEVATE		0.97	0.62	0.49	0.54	0.52
HPVM2FPGA	BFS	1.00	0.34	0.70	0.56	0.50
HPVM2FPGA	Audio	1.00	0.90	0.91	0.86	0.83
HPVM2FPGA	PreEuler	0.78	0.45	0.36	0.50	0.48
HPVM2FPGA		0.92	0.57	0.66	0.64	0.61
All		1.22	0.80	0.61	0.67	0.71

Table 7: Relative performance compared to expert with the small budget (2/3 of the full budget). Values larger than 1 indicate a performance advantage over the expert. Values below 1 indicate a performance disadvantage compared with the expert.

Framework	Benchmark	BaCO	ATF	Ytopt	Uniform	CoT
TACO (SpMM)	scircuit	1.31	1.12	0.85	0.90	0.91
TACO (SpMM)	cage12	1.37	1.08	0.82	0.90	0.92
TACO (SpMM)	laminarduct3D	1.24	0.77	0.48	0.52	0.63
TACO (SDDMM)	email-Enron	1.08	1.04	0.96	0.95	1.00
TACO (SDDMM)	ACTIVSg10K	1.04	1.03	0.98	0.96	0.99
TACO (SDDMM)	Goodwin040	1.09	1.05	0.96	0.95	0.98
TACO (MTTKRP)	uber	1.00	0.64	0.24	0.38	0.54
TACO (MTTKRP)	nips	1.00	1.00	0.81	0.78	0.91
TACO (MTTKRP)	chicago	1.00	1.05	0.88	0.61	0.80
TACO (TTV)	facebook	2.52	1.90	0.65	1.25	1.16
TACO (TTV)	uber3	1.00	0.75	0.61	0.69	0.62
TACO (TTV)	random	8.82	3.32	1.81	2.32	2.90
TACO (SpMV)	laminarduct3D	0.98	0.91	0.54	0.60	0.68
TACO (SpMV)	cage12	0.77	1.48	0.47	0.63	0.62
TACO		1.66	1.24	0.77	0.86	0.95
RISE & ELEVATE	MM-CPU	1.02	0.93	0.88	0.83	0.83
RISE & ELEVATE	MM-GPU	1.05	0.49	0.49	0.44	0.29
RISE & ELEVATE	Asum-GPU	1.25	0.67	0.43	0.50	0.58
RISE & ELEVATE	Scal-GPU	1.08	0.14	0.23	0.23	0.14
RISE & ELEVATE	K-means-GPU	1.02	0.84	0.40	0.54	0.79
RISE & ELEVATE	Stencil-GPU	0.98	0.91	0.85	0.84	0.82
RISE & ELEVATE		1.06	0.69	0.55	0.61	0.59
HPVM2FPGA	BFS	1.00	0.45	0.74	0.69	0.67
HPVM2FPGA	Audio	1.00	0.94	0.94	0.90	0.91
HPVM2FPGA	PreEuler	1.00	0.50	0.51	0.57	0.56
HPVM2FPGA		1.00	0.63	0.73	0.72	0.71
All		1.41	1.01	0.71	0.76	0.83

Table 8: Relative performance compared to expert with the full budget. Values larger than 1 indicate a performance advantage over the expert. Values below 1 indicate a performance disadvantage compared with the expert.

Framework	Benchmark	ATF	Ytopt	Uniform	CoT
TACO (SpMM)	scircuit	3.33×	5.00×	4.62×	4.62×
TACO (SpMM)	cage12	3.53×	6.00×	5.45×	5.00×
TACO (SpMM)	laminarduct3D	4.00×	5.45×	5.00×	4.29×
TACO (SDDMM)	email-Enron	3.75×	4.62×	4.62×	4.29×
TACO (SDDMM)	ACTIVSg10K	1.88×	4.62×	5.45×	3.75×
TACO (SDDMM)	Goodwin040	2.40×	4.29×	4.62×	4.00×
TACO (MTTKRP)	uber	1.40×	4.62×	1.88×	1.43×
TACO (MTTKRP)	nips	1.03×	2.31×	2.40×	1.58×
TACO (MTTKRP)	chicago	-	2.14×	3.53×	2.86×
TACO (TTV)	facebook	1.76×	3.00×	2.40×	2.50×
TACO (TTV)	uber3	2.86×	3.53×	3.16×	3.33×
TACO (TTV)	random	1.67×	2.07×	2.07×	1.67×
TACO (SpMV)	laminarduct3D	1.09×	3.33×	2.86×	2.73×
TACO (SpMV)	cage12	-	4.00×	1.76×	1.76×
TACO		3.15×	5.00×	4.96×	4.32×
RISE & ELEVATE	MM-CPU	2.22×	3.03×	3.85×	3.85×
RISE & ELEVATE	MM-GPU	2.03×	1.97×	2.18×	2.93×
RISE & ELEVATE	Asum-GPU	2.73×	3.53×	3.33×	3.00×
RISE & ELEVATE	Scal-GPU	4.62×	3.75×	3.75×	4.62×
RISE & ELEVATE	K-means-GPU	2.73×	6.00×	5.00×	3.16×
RISE & ELEVATE	Stencil-GPU	2.50×	3.53×	3.53×	3.75×
RISE & ELEVATE		2.68×	3.38×	3.51×	3.55×
HPVM2FPGA	BFS	2.86×	2.00×	2.22×	2.22×
HPVM2FPGA	Audio	2.86×	2.86×	3.16×	3.00×
HPMV2FPGA	PreEuler	2.61×	2.61×	2.00×	2.07×
HPVM2FPGA		2.77×	2.49×	2.46×	2.43×
all		2.87×	3.82×	3.86×	3.63×

Table 9: Factors showing how much faster BaCO reach the best performance of the other methods. E.g., a factor of 3.33× indicates that BaCO required 3.33× less evaluations to achieve the same performance. "-" indicates that the average final performance of BaCO was lower than ATF.

B WALL CLOCK TIME ANALYSIS

In Table 10, we show the average wall clock time of BaCO and the baselines for the TACO SpMM and SDDMM benchmarks. The breakdown of the wall clock time is given by two main factors: 1) the evaluations of the black-box function, and 2) the computing time for the autotuner to generate its recommendations. The most expensive part of the autotuning process is the first factor, where the program kernel is evaluated a number of times corresponding to the budget allocated. Thus, methods that recommend slow-to-evaluate configurations tend to use more wall clock time per evaluation. The second factor relates to the internal workings of an autotuner search method. In this regard, the more intricate model-based methods tend to be slower, but this cost becomes less prominent when optimizing larger program kernels. This factor is highly dependent on the quality of the software implementation of the autotuner. It is beyond the scope of this work to provide the fastest implementation of both BaCO and the baselines — In Table 10 we give a rough analysis of the current average wall clock time for all autotuners, which will likely be improved in future releases of these tools. We observe that while BaCO uses a more complex method than the baselines it is the second fastest method behind ATF. ATF uses OpenTuner which employs search algorithms based on heuristics — These are usually faster than model-based approaches such as BaCO. We can see this insight reflected also in the Ytopt wall clock time.

	BaCO	ATF w. OpenTuner	Ytopt	Uniform sampling	CoT samp.
SpMM	262	**144**	309	402	336
SDDMM	263	**197**	274	433	381

Table 10: Average wall clock time in seconds for BaCO and the baselines on the TACO SpMM and SDDMM benchmarks.

C ARTIFACT APPENDIX

C.1 Abstract

This artifact appendix describes how to install BaCO using docker containers together with the TACO and RISE/ELEVATE benchmark suites. For a custom installation, we refer to https://github.com/baco-authors/baco/. Further, this appendix describes how to run the experiments described in the paper. We have decided to focus on the two major benchmark suites TACO and RISE/ELEVATE, as the simulator used in HPVM2FPGA uses 70GB of installation space and generates around 300GB of auxiliary data each run. The experiments can be run on any linux distribution with Docker, git and bash support. The experiments will require around 50GB of free space and are runnable on a medium to high-end personal computer.

Running the fire tests requires around 10 hours of computational time, while running all experiments presented in the paper takes up towards 500 hours. However, the paper was run with a high number of repetitions for statistical significance; similar trends should be visible already at significantly lower repetition count.

C.2 Artifact check-list (meta-information)

- **Algorithm:** We present BaCO, a new framework for Autotuning with Bayesian Optimization
- **Data set:** TACO uses matrices and tensors from the Suitesparse (https://sparse.tamu.edu/) and Frostt (http://frostt.io/) repositories.
- **Run-time environment:** Docker, git, and bash are required. Proficiency in Docker and git would be helpful.
- **Hardware:** Most high-end personal computers with a graphics card should suffice for the fire test. To recreate the full set of results, some form of additional computational resources would be necessary.
- **Metrics:** The metric considered in this work is the final runtimes of the optimized kernels. Especially, we consider speedup over expert configurations as a target.
- **Output:** .csv data files, log files and .pdf figures
- **Experiments:** The experiments show how one can learn good tuning configuration faster by using efficient optimization algorithm
- **How much disk space required (approximately)?:** Less than 50 GB
- **How much time is needed to prepare workflow (approximately)?:** Less than an hour of manual time
- **How much time is needed to complete experiments (approximately)?:** The complete runtime of rerunning all experiments is around 400 hours. We provide a fire test that runs in approximately 10 hours.
- **Publicly available?:** The code is publically available at https://github.com/baco-authors/baco/ and the artifact at https://github.com/baco-authors/baco-artifact/.
- **Code licenses (if publicly available)?:** MIT-licence
- **Archived (provide DOI)?:** The artifact as well as the BACO repo are archived at zenodo with DOI: 10.5281/zenodo.10116240

C.3 Description

C.3.1 How to access. The files for this artifact can be found at https://github.com/baco-authors/baco-artifact.git. The repository contains three separate docker files, one each for the two benchmarks and one for the plotting script.

C.3.2 Hardware dependencies. The benchmarks can be run on a standard personal computers, but the behaviour of the benchmarks are dependent on the hardware, so the results might change. In particular, TACO considers the number of threads to use as a tunable parameter. The experiments presented in the paper were run with 32 cores. Using less than that would reduce the size of feasible space and lessen the impact of that particular variable. For RISE/ELEVATE, many of the applications were intended to be run on a GPU. If the setup does not have access to a GPU, we provide a CPU version, but again that can have a significant impact on the behaviour of the benchmarks.

In terms of memory consumption, the artifact evaluation requires less than 50GB disk space. Some of the tensors used in TACO are quite large, and, if possible, running it with 32GB of RAM is preferable.

However, in terms of runtime, running it on a personal computer will be insufficient to finish the complete set of experiments in a reasonable time. Our estimate is that the complete runtime to run 30 iteration of each method would be yp towards 500 hours.

C.3.3 Software dependencies. The artifact uses the developed BaCO framework together with TACO and RISE/ELEVATE. It also uses the baselines, Ytopt and OpenTuner. Each of this is automatically installed from the Dockerfiles, and as such, the user only needs *bash*, *git*, and *docker* preinstalled locally.

C.3.4 Data sets. TACO uses matrices and tensors from the SuiteSparse (https://sparse.tamu.edu/) and the Frostt (http://frostt.io/)) data repositories. Those will be automatically downloaded during installation. In total, the tensors take up a few GB of disk space. In addition to this, we provide the raw data files from the original manuscript at the Zenodo repository with DOI: 10.5281/zenodo.10116240.

C.4 Installation

Manual time: 2 min
Compute time: 60 min

In this section we will step by step describe how to install the artifact. The installation instructions will be further detailed in the README of the repo https://github.com/baco-authors/baco-artifact.git.

First, clone the repo

```
$ git clone https://github.com/baco-authors/baco-artifact
$ cd baco-artifact
```

then run the install script

```
$ ./install_all.sh
```

This builds the three docker images and creates two empty folders "results" and "plots". The containers are named "*taco*", "*rise*" and "*plot*".

Each docker image installs BaCO and the baselines from https://github.com/baco-authors/baco and links it with the corresponding benchmark.

C.5 Experiment workflow

Manual time: 20 min
Compute time: 10 hours

After installing the containers, a predefined set of experiments are exposed through *running* the container. Here one has the option of how many repetitions to perform, trading computation time

for statistical reliability. Note that the run commands uses local bindings and MUST be run from the baco-artifact root directory. For TACO, a single run is performed by the following command.

```
$ docker run -it -v \\
    "$(pwd)"/results/taco:/home/taco/build/experiments taco 1
```

For RISE/ELEVATE it is instead run interactively.

```
$ docker run --gpus all -it -v \\
    "$(pwd)"/results:/home/shine/artifact/results rise bash
$ cd /home/shine
$ ./run_rise.sh 1
$ ./run_ablation.sh 1
```

This enters the run directory, and runs 1 iterations each of the main benchmarks and then the ablation experiments.

Once the data is run, the results can be plotted using the plot container.

```
$ docker run -v "$(pwd)"/results:/app/results \\
    -v "$(pwd)"/plots:/app/plots plot
```

It will read the results from the local results folder and store them in the local plots folder.

C.6 Evaluation and expected results

Manual time: 10 minutes

With all the experiments run, and the plots made, we expect to see similar trends as in the papers. The actual runtimes and the behaviour of the benchmarks will be different depending on the hardware it is run on, but the general trends, such as large improvements and better-than-expert-performance are still expected. We expect the order between the methods to stay roughly the same, and time until convergence too. However, the amplitude of the changes and the actual values will depend. For certain hardware, the default configuration will work better or worse, and that drastically changes how much potential there is for optimization.

We also provide the original at DOI: 10.5281/zenodo.10116240. This can be copied into the local results folder in the artifact and be plotted using the plot image.

C.7 Experiment customization

The main purpose of this artifact is to reproduce the results of the paper. If one wants to experiment further and use BaCO in their own applications, we instead refer to the BaCO repo, https://github.com/baco-authors/baco. BaCO is under constant development and this lets the user stay up to date with new innovations. Yet, the artifact experiments can easily be customized. Just change the run files to include the experiments to run. BaCO uses a .json settings file, that exposes a large possibility for customizing the optimizer from the outside. More instructions can be found in the BaCO repository.

C.8 Notes

Due to the massive memory consumption of HPVM2FPGA, we have opted to not include it in this artifact evaluation. It is also the least significant of the three benchmark sets. First of all, we have only run three benchmarks on it due to said constraints. Secondly, it uses a simulator instead of measuring the real performance of a physical system. It is our belief that the trends shown in this paper can be well verified on the 20 benchmark instances provided in this artifact.

C.9 Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html