Accuracy-aware mixed-precision GPU auto-tuning

Heldens, S.; Werkhoven, B.J.C. van

Citation
Heldens, S., & Werkhoven, B. J. C. van. (2026). Accuracy-aware mixed-precision GPU auto-tuning. *Ieee Transactions On Parallel And Distributed Systems*, *37*(4). doi:10.1109/TPDS.2026.3659324

Version: Publisher's Version
License: Creative Commons CC BY 4.0 license
Downloaded from: https://hdl.handle.net/1887/4297173

Note: To cite this publication please use the final published version (if applicable).

Accuracy-Aware Mixed-Precision GPU Auto-Tuning

Stijn Heldens and Ben van Werkhoven

***Abstract*—Reduced-precision floating-point arithmetic has become increasingly important in GPU applications for AI and HPC, as it can deliver substantial speedups while reducing energy consumption and memory footprint. However, choosing the appropriate data formats brings a challenging tuning problem: precision parameters must be chosen to maximize performance while preserving numerical accuracy. At the same time, GPU kernels typically expose additional tunable optimization parameters, such as block size, tiling strategy, and vector width. The combination of these two kinds of parameters results in a complex trade-off between accuracy and performance, making manual exploration of the resulting design space time-consuming. In this work, we present an *accuracy-aware* extension to the open-source *Kernel Tuner* framework, enabling automatic tuning of floating-point precision parameters alongside conventional code-optimization parameters. We evaluate our accuracy-aware tuning solution on both Nvidia and AMD GPUs using a variety of kernels. Our results show speedups of up to 12× over double precision, demonstrate how Kernel Tuner's built-in search strategies are effective for accuracy-aware tuning, and show that our approach can be extended to other optimization objectives, such as memory footprint or energy efficiency. Moreover, we highlight that jointly tuning accuracy- and performance-affecting parameters outperforms isolated approaches in finding the best-performing configurations, despite significantly expanding the optimization space. This unified approach enables developers to trade accuracy for throughput systematically, enabling broader adoption of mixed-precision computing in scientific and industrial applications.**

***Index Terms*—GPU computing, mixed-precision, reduced-precision floating point, automatic performance optimization, auto-tuning, high-performance computing (HPC), energy-efficient computing.**

I. Introduction

Advances in Graphics Processing Unit (GPU) technology have greatly accelerated large-scale AI and High-Performance Computing (HPC) applications [1], [2]. In particular, reduced-precision floating-point data formats, ranging from 16-bit to even lower-precision formats [3], [4], [5], [6], have received significant attention for their potential to increase computational throughput, reduce memory footprint, and lower energy consumption. Motivated by their success in deep learning, GPU vendors have introduced specialized hardware (e.g., Tensor Cores [7] and Matrix Cores [8]) specifically designed for low-precision operations. This has allowed for computational throughput well beyond traditional single- (32 b) and double-precision (64 b) floating-point arithmetic.

While originally targeted at machine learning, these low-precision formats can also offer substantial opportunities for scientific computing. Unfortunately, integrating lower precision into HPC workloads poses several challenges. For instance, using a single low-precision format throughout a code typically leads to an unacceptable loss in accuracy, so one must mix different levels of precision within the application. However, where to apply lower precision is problem- and data-dependent; an understanding of numerical sensitivity is required as there is no one-size-fits-all solution.

GPU programming models, such as HIP, CUDA, and OpenCL, allow developers to create highly parallel functions, called *kernels*, that are executed on the GPU by millions of threads in parallel. When writing such kernels, GPU programmers are confronted with many implementation decisions related to thread organization, memory access patterns, and algorithmic choices that affect computational throughput. A recent survey by Hijma et al. [9] presents a comprehensive overview of the wide variety of programming techniques used to improve the performance of GPU applications in the literature. Many of these programming techniques introduce tunable parameters that in turn need to be configured to achieve optimal performance, such as thread-block size, tile dimensions, vector widths, and loop-unrolling factors.

The performance improvements gained by switching to low-precision formats are not uniform and depend on these optimization-related parameters. For example, for tiling, the tile size and floating-point formats are related as lowering the precision could reduce register usage, which frees up resources that can be used to increase the tile size. In other words, making precision or optimization decisions in isolation leads to suboptimal solutions: finding the best configuration often requires a unified approach that simultaneously tunes *both* accuracy- and optimization-relevant parameters.

However, tuning these parameters jointly greatly expands the optimization space. This raises the research question: *How will unified tuning impact the performance of optimization algorithms and will a unified approach actually outperform isolated optimization approaches?* To investigate the impact and challenges of unified performance and precision tuning, we extend an existing automatic performance tuning framework, *Kernel Tuner* [10], with accuracy tuning capabilities. Kernel Tuner is a widely used open-source[1] auto-tuning tool that enables

Received 18 July 2025; revised 13 January 2026; accepted 14 January 2026. Date of publication 29 January 2026; date of current version 23 February 2026. This work was supported in part by CORTEX Project, funded by Dutch Research Council (NWO) under NWAORC Call file number NWA.1160.18.316 and in part by ESiWACE3 Project, funded by EuroHPC Joint Undertaking (JU) and national co-funding bodies under Grant Agreement 101093054. Recommended for acceptance by F. Zhang. *(Corresponding author: Stijn Heldens.)*

Stijn Heldens is with Netherlands eScience Center, 1098 XH Amsterdam, The Netherlands (e-mail: s.heldens@esciencecenter.nl).

Ben van Werkhoven is with Leiden University, 2311 EZ Leiden, The Netherlands.

Digital Object Identifier 10.1109/TPDS.2026.3659324

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

© 2026 The Authors. This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

developers to automatically search for the best combination of performance-affecting parameters for GPU kernels. Our work extends Kernel Tuner by enabling exploration of floating-point precisions in combination with traditional code-tuning parameters. Specifically, it integrates *mixed-precision* into the search process, treating accuracy-affecting choices (like the floating-point formats) as tunable parameters, just like traditional optimization parameters. By measuring numerical error and performance metrics together, we enable developers to explore the trade-off between numerical accuracy and computational performance.

We demonstrate how this accuracy-aware tuning approach yields significant performance improvements, sometimes by more than an order of magnitude over 64-bit baselines, for a diverse set of GPU kernels, while still providing acceptable numerical accuracy. Moreover, our results show that the relation between new precision parameters and traditional optimization parameters is rarely straightforward: tuning one aspect (e.g., the data type of a local variable) can affect the optimal setting of another (e.g., vector width), underscoring the need for a unified approach. Our extension to Kernel Tuner simplifies the tuning process by providing search strategies that automatically explore this search space.

In summary, the main contributions of this work are:

- An extension to *Kernel Tuner* that enables joint tuning of floating-point formats together with conventional GPU parameters in CUDA/HIP kernels.
- A method for measuring numerical accuracy during tuning, enabling *accuracy-aware* auto-tuning.
- Benchmarks on Nvidia and AMD GPUs, demonstrating practical benefits from accuracy-aware tuning, including performance improvements (up to $12\times$), memory footprint reductions (up to $4\times$), and increases in energy efficiency (up to $3.8\times$).
- Demonstration that Kernel Tuner's built-in search strategies are effective for accuracy-aware tuning, with the best strategy outperforming random sampling in 90% of cases.
- An analysis of precision–performance interdependence, showing that sequentially tuning precision and performance parameters, in either order, achieves only 77% or 86% of the performance compared to 96% of our unified tuning approach.

The rest of this paper is structured as follows. Section II discusses background and related work. Section III provides an introduction to automatic performance tuning using Kernel Tuner and presents our extensions. Section V evaluates the impact of mixed-precision auto-tuning on both performance and accuracy across a wide range of benchmark applications. Finally, Section VI concludes this work.

II. Background & Related Work

This section presents an overview of related work on optimizing GPU applications for performance (Section II-A), for precision (Section II-B), and for a combination of both (Section II-C). Finally, Section II-D provides an introduction to metaprogramming for mixed-precision GPU kernels.

A. Generic Auto-Tuning Frameworks

Automatic performance tuning, or *auto-tuning*, enables developers to optimize software efficiently for specific hardware and input configurations [11], [12]. Generic auto-tuning frameworks provide an application-independent approach for users to create tunable applications, in which performance-critical parameters (e.g., the number of threads, work per thread, and data layouts) can be varied [10], [13], [14], [15].

An auto-tuner constructs a search space $\mathcal{X}$ based on all combinations of these *tunable* parameters in the code that satisfy user-defined constraints. If we let $f(x)$ be the optimization objective (e.g., computational performance or the energy efficiency) of a configuration of tunable parameters $x \in \mathcal{X}$, we can treat the auto-tuning problem as a numerical optimization problem:

$$x^{\star} = \arg\max_{x \in \mathcal{X}} f(x). \quad (1)$$

Each x represents a variant of the user-provided code that is typically generated using metaprogramming or compilation techniques. Note that, while each code variant x computes the same output, its performance $f(x)$ can vary dramatically. Evaluating $f(x)$ is expensive because it requires compiling, executing, and benchmarking the code variant on the hardware. Auto-tuners systematically explore the search space to automatically identify the parameter configurations that maximize performance [9], [16], [17] or energy efficiency [18]. In this paper, we focus on *compile-time* auto-tuning where the application can be tuned as part of the compilation process, as opposed to *run-time* auto-tuning where the application is tuned while it is running in production [19].

There are several generic open-source auto-tuning frameworks, including OpenTuner [13], CLTune [14], KTT [19], ATF [27], pyATF [28], GPTune [29], ytopt [30], HyperMapper [31] and BaCO [32], and Kernel Tuner [10]. These frameworks generally target different aspects of auto-tuning and are largely complementary approaches. For example, while OpenTuner and ytopt are designed as libraries that allow developers to implement their own auto-tuners, ATF, pyATF, KTT and Kernel Tuner provide a high-level user interface where the user only specifies their kernel code and tunable parameters, and the tuner automatically takes care of compiling and measuring performance. While GPTune, ytopt, HyperMapper and BaCO all use Bayesian Optimization to navigate the search space, other frameworks such as OpenTuner, CLTune, KTT, ATF, and pyATF support multiple optimization algorithms. In particular Kernel Tuner supports a wide range of different optimization algorithms, including Bayesian Optimization [33]. However, despite the great diversity in available frameworks, none of the existing generic auto-tuning frameworks has built-in support for precision tuning, such as allowing users to specify tunable parameters that influence precision or recording the numerical error of different code variants along with performance or other characteristics.

TABLE I
OVERVIEW OF COMPILER-BASED PRECISION-TUNING FRAMEWORKS FOR GPU APPLICATIONS

Name	Year	Supported FP formats	Tuning target	Compiler	Target language	Optimization algorithm
Nobre et al. [20]	2018	FP16, FP32, FP64	variables	Clava compiler	OpenCL (LARA)	exhaustive search
AMPT-GA [21]	2019	FP16, FP32, FP64	variables	LLVM	CUDA	genetic algorithm
GPUMixer [22]	2019	FP32, FP64	sets of operators	Clang (NVVM IR)	CUDA	exhaustive search
GPU-FPTuner [23]	2020	FP32, FP64, FP128	variables	ROSE	CUDA	iterative search
PyFloT [24]	2020	FP32, FP64	math functions	link-time interposition	various	exhaustive search
GRAM [25]	2021	FP16, FP32, FP64	various	Clang (LLVM)	CUDA	binary search
MoTuner [26]	2022	8-bit int, FP16, FP32	operators	LLVM	HIP	optimized adjustment

We have chosen Kernel Tuner [10] because it already provides an easy to extend API for creating new programmable hooks, called *observers* in Kernel Tuner [18], and because it implements a great number of optimization algorithms to efficiently navigate the auto-tuning search space.

B. Precision-Tuning Frameworks

In the past two decades, several precision tuners have been proposed that automatically quantify the numerical error that would be introduced when certain variables [34], [35], [36], [37] or operations [38] are moved to lower precision without empirical measurements. Recently, Wang and Rubio-González [39] have used graph neural networks to predict the accuracy of four mixed-precision programs. These approaches typically assume that lower precision means higher performance, which, as we will show in this paper, is not always true. Moreover, these approaches have not been developed for mixed-precision computing on GPUs.

It is interesting to note that most approaches for mixed-precision tuning of GPU applications are compiler-based [20], [21], [22], [23], [24], [25], [26]. This means the user inputs an application to a custom compiler, which automatically optimizes the application based on some user-defined error threshold. In many cases the input application is first modified, instrumented and/or profiled to let the compiler framework gather information that cannot be derived from static analysis.

Table I presents an overview of compiler-based mixed-precision auto-tuners for GPU applications. Nobre et al. [20] present a compiler-based approach using the Clava source-to-source compiler to generate different versions of OpenCL applications that have been instrumented using the LARA annotation language. AMPT-GA [21] use a genetic algorithm to search among different combinations of precision levels for GPU kernel inputs, outputs, and local variables. GPUMixer [22] uses static analysis to identify sets of operators that can be moved as a group to lower precision. GPU-FPTuner [23] focuses on extending precision to 128 bits instead of reduced precision computing. GPU-FPTuner does take an interesting approach that targets whole applications and also takes CPU-GPU data transfers into account, detects accumulation patterns within kernels, and variable use across multiple GPU kernels. We believe this approach is complementary to our work, which focuses optimizing individual kernels. In contrast, PyFloT [24] only targets individual calls to mathematical functions, for which it can replace the double precision version at link time with a single precision version. PyFloT only supports single and double precision. MoTuner [26] is a compiler-based approach for lowering precision of operators supporting 8-bit integers, half, and single precision.

C. Unified Performance-Precision Tuning

Out of the precision tuners presented in Table I, the approach taken by GRAM [25] is the most closely related to ours, as it supports automatic vectorization of kernels (from `half` precision to `half2`), which is why we classify it here as a unified performance-precision tuning approach. In contrast to the frameworks presented in Section II-A, GRAM cannot be used to tune any other user provided tunable parameters. Instead, GRAM implements a limited number of techniques to create mixed-precision implementations of GPU kernels. GRAM has the ability to assign different thread blocks within a single kernel to different precision levels at runtime. Thus, instead of searching for a subset of certain *variables* to lower precision levels to gain more performance, GRAM instead assigns different *data elements* to lower precision levels. In this way, GRAM allows combining FP16 and FP32, as well as FP32 and FP64. However, while being a compiler-based framework, GRAM requires the user to manually write the kernel code in different precision levels and provide them to the framework, which are then combined into a single kernel that satisfies a user-specified error threshold.

In contrast to the literature, which predominantly consists of compiler-based frameworks, our paper presents an orthogonal approach. Instead of using a modified compiler to generate code variants based on a single source code, our approach relies on standard compilers and instead uses *metaprogramming* (e.g. C++ templates or C preprocessor defines) to specify a code template with configurable precision levels.

D. Metaprogramming for Mixed-Precision GPU Kernels

Mixed-precision GPU programming introduces several challenges. For instance, using 16-bit precision requires special data types and low-level compiler intrinsics (for example, `__hadd2(x, y)` performs vectorized addition of two 16-bit pairs). The available data types and intrinsics also differ between GPU vendors (AMD and Nvidia) and even across GPU generations, making it challenging to write portable code with configurable data types.

```
1  #include "kernel_float.h"
2  namespace kf = kernel_float;
3
4  __global__ void axpy(
5    kf::vec_ptr<YTYPE, VECTOR_SIZE> y,
6    kf::vec_ptr<const XTYPE, VECTOR_SIZE> x,
7    ATYPE a,
8    const int n)
9  {
10   int i = BLOCK_SIZE_X * blockIdx.x + threadIdx.x;
11   if (i < n/VECTOR_SIZE) {
12     y[i] += a * x[i];
13   }
14 }
```

Listing 1: Example of CUDA/HIP kernel using Kernel Float.

Kernel Float [40] is a recently developed open-source[2] C++ header-only library that simplifies development of mixed-precision GPU applications for CUDA (Nvidia GPUs) and HIP (AMD GPUs). Kernel Float provides a generic vectorized data type and by using operator overloading, Kernel Float inserts type casts where required and automatically selects the optimal intrinsic function needed by the operation and data type at hand. Kernel Float makes it trivial to change the vector width or scalar type of variables. As such, it is no longer needed to rely on customized compilers to generate code variants with different mixes of precision levels.

Listing 1 shows an example of the `axpy` kernel implemented in HIP/CUDA using Kernel Float. This kernel takes a scalar a and vectors x, y and computes $y := ax + y$, a common linear algebra operation and part of the BLAS routines. This kernel code is essentially a template that can be instantiated to use different types and vector lengths. Specifically, the constants `ATYPE`, `XTYPE`, and `YTYPE` can be used to select the floating-point format for the variables `a`, `x` and `y`, respectively. The format could be set to any data type (e.g., `double` or `half`), without further code modifications. Meanwhile, the constants `BLOCK_SIZE_X` and `VECTOR_SIZE` control the thread block dimensions (i.e., number of threads per block) and vector width (i.e., number of elements per thread).

By relying on Kernel Float's portability, we are able to use mainstream GPU compilers (e.g., `hiprtc` or `nvrtc`). Thus, we can use a generic source-code-level auto-tuner extended with mixed-precision tuning capabilities. This removes the need for custom compiler setups that may not be available to end-users of applications. The main advantage of a compiler-based approach would be that the input application does not need to be modified by the programmer. However, as we have seen in Section II-B and II-C many compiler-based approaches still require the user to supply multiple different versions or instrumented versions of the application [20], [25].

III. An Auto-Tuner for Mixed-Precision Kernels

In this section, we introduces Kernel Tuner (Section III-A) and presents our two main extensions: support for variable-typed kernel arguments (Section III-B) and computation of numerical accuracy with a user-defined error metric (Section III-C). Finally, we explain how to use Kernel Tuner's user-defined metrics for accuracy-aware optimization (Section III-D) and we show a simple code example (Section III-F).

[2]https://github.com/KernelTuner/kernel_float

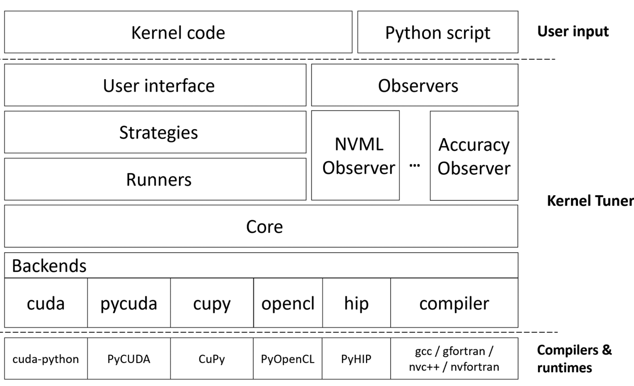

Fig. 1. Kernel Tuner software architecture.

A. Kernel Tuner

Kernel Tuner [10], [41] is a Python-based open-source auto-tuner for GPU applications in HIP, OpenCL, and CUDA that can easily be extended. Its layered software architecture, including our new contributions, is shown in Fig. 1. The user calls Kernel Tuner from Python with a description of the GPU code to be tuned. The *strategies* layer implements a great variety of optimization algorithms, selecting code variants for evaluation by the *runner*. The runner interacts with the core layer, which in turn abstracts a diverse set of supported compilers and hardware by using one of the supported *backends*. The backends offer the device-specific functionality supplied by HIP, OpenCL, CUDA, and other compilers into a uniform interface. This separation lets highlevel components (e.g., runners, optimization strategies) remain agnostic to the underlying programming model and hardware. Kernel Tuner provides an *observer* interface to perform measurements before, after, or during the benchmarking of each code variant. To support tuning the data type of kernel arguments, we extended the core layer (Section III-B). To measure the numerical error, we introduced a new type of *observer* (Section III-C).

B. Tuning Data Types of Kernel Inputs/Outputs

The data types of the kernel arguments are crucial factors influencing both computational performance and numerical accuracy. Changing the data type of one of the kernel arguments to a type of lower precision can offer several important advantages in terms of performance, as it can reduce memory bandwidth and improve cache utilization.

Kernel Tuner automates compiling and benchmarking kernels on GPUs, without the need to write separate host code for testing individual functions. Kernel Tuner does this by inferring the kernel input and output data types and dimensions from the data passed to the tuner. Using this information, Kernel Tuner allocates the required buffers and copies data into and

out of GPU memory as needed. However, when the types of the kernel's input and output data become tunable, the tuner needs to specifically support this functionality.

To this end, we have introduced a new concept in Kernel Tuner, called `TunablePrecision`, which allows specifying that the input or output argument of a kernel has a data type determined by a tunable parameter. When wrapping a kernel argument as a `TunablePrecision` object, Kernel Tuner creates different versions of the data for each required precision. When the kernel is compiled and benchmarked on the GPU, Kernel Tuner ensures that the correct version of the kernel argument is present in GPU memory and is passed to the kernel. This allows the tuner to evaluate the impact on both performance and accuracy when varying the data types of kernel arguments.

C. Accuracy Measurements

Kernel Tuner offers an *observer* mechanism as a flexible way to extend the measurement capabilities beyond only execution time. Named after the observer programming pattern, this mechanism serves as a flexible way for end-users to add custom measurements without the need to modify Kernel Tuner's source code. For instance, observers can be used to also record other quantities, including the number of registers required, performance counter data collected by a profiler, or power management statistics such as GPU temperature, memory clock and core clock frequencies, core voltage, and GPU power consumption [18].

To measure the numerical accuracy of a specific code variant, we have introduced a new type of observer in Kernel Tuner called the `AccuracyObserver`. This new observer quantifies the error in the output data produced by the kernel executed on the GPU by comparing it to user-provided reference data. Calculating the error requires a metric and the choice of an appropriate error metric depends on the application at hand. The `AccuracyObserver` supports several built-in options (e.g., mean relative error, root-mean-squared error, and mean absolute error) and custom metrics.

For each code variant, Kernel Tuner will perform several steps. First, the kernel is compiled and executed once on the GPU. Next, the resulting output data is copied to the host and passed to each `AccuracyObserver`, which compute the error metric. Finally, the kernel is benchmarked for performance, Kernel Tuner gathers all observed measurements, and computes any user-specified derived metrics.

D. Accuracy-Aware Objective Function

Besides the measurements collected by observers, Kernel Tuner also lets users define custom *derived* metrics. For example, given a kernel's execution time in milliseconds, one can define a metric that reports the achieved throughput in GFLOP/s (billions of floating-point operations per second).

We use this mechanism to define an accuracy-aware objective function $f(x)$ to be maximized (see (1), Section II-A). For accuracy-aware tuning, the goal is to maximize a chosen performance metric (e.g., speedup, memory bandwidth or compute throughput) while ensuring that the numerical error remains below a user-specified threshold τ. For example, if the error is defined as average number of correct decimal places, then $\tau = 5$ means that, on average, the first five decimal places should be correct. Therefore, $f(x)$ returns the performance of configuration x when $\text{error}_x < \tau$; otherwise it returns a penalty:

$$f(x) = \begin{cases} \text{performance}_x & \text{if error}_x < \tau, \\ \text{penalty}(x) & \text{otherwise.} \end{cases}$$

The penalty function determines how a search strategy behaves once it enters a region of the search space that fails to meet the accuracy requirement. Defining penalty functions is an active area of research [42]. The specific performance metric, error metric, and penalty function should be chosen by the user as the optimal choice is application-dependent.

E. Search Strategies

When tuning, the size of the search space (i.e., the Cartesian product of the parameters) can be so large that an exhaustive search is impractical. For example, for one of the kernels from our evaluation (Convolution2D), even when evaluating one configuration per second, a full sweep of the search space would take more than 20 hours. Fortunately, Kernel Tuner provides several built-in *search strategies* that use stochastic methods to efficiently explore the space and identify near-optimal configurations by only taking samples.

Each configuration in Kernel Tuner is represented as an N-dimensional point, where N is the number of tunable parameters. To allow continuous optimization algorithms, such as Particle Swarm Optimization, to work on the discrete optimization problem of auto-tuning, Kernel Tuner embeds the discrete values (e.g. `double`, `float`, and `half`) linearly into a continuous domain $[0, 1]^N$. For each parameter i, which has m_i discrete values, the values are mapped to uniformly spaced points $0.5\epsilon, 1.5\epsilon, \ldots, (m_i - 0.5)\epsilon$. The value ϵ is computed as $\frac{1}{m_{\text{max}}}$, where m_{max} is the maximum of all m_i. This method ensures that regardless of the number of values in a particular dimension, a perturbation by ϵ in any dimension leads to a position that represents a different solution in the discrete space. This numeric representation enables all existing search algorithms to be used for precision tuning directly, without modifying their implementation.

Below, we include a brief description of each method. For detailed descriptions, we refer to the following work: for BH, PSO, and FF see [10]; for SA, GA, and MLS see [43]; and for BO see [33]. The source code of Kernel Tuner and its search strategies is available as open source software[3].

Rand Randomly samples valid configurations from the search space until the optimization budget is exhausted.

GA Genetic Algorithm uses a population of candidate solutions that is completely refreshed every generation. Individuals are sorted based on the objective function value and selected following a beta probability distribution to ensure individuals with higher objective score have a higher probability of being selected, independently of the magnitude of the

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

```
1  import numpy as np
2  from kernel_tuner import tune_kernel
3  from kernel_tuner.accuracy import TunablePrecision, AccuracyObserver
4
5  n = np.int32(120000000)
6  a = np.float64(np.random.rand())
7  x = np.random.randn(n).astype(np.float64)
8  y = np.random.randn(n).astype(np.float64)
9
10 args = [TunablePrecision("YTYPE", y), TunablePrecision("XTYPE", x), TunablePrecision("ATYPE", a), n]
11
12 answer = [y+a*x, None, None, None]
13 observers = [AccuracyObserver("RMSE", "error_rmse")]
14
15 metrics = dict()
16 metrics["GFLOP/s"] = lambda p: (2*n / 1e9) / (p["time"] / 1e3)
17 metrics["fitness"] = lambda p: p["GFLOP/s"] if np.log10(p["error_rmse"]) < -7 else 0
18
19 tune_params = dict()
20 tune_params["BLOCK_SIZE_X"] = [32, 64, 128, 256, 512, 1024]
21 tune_params["YTYPE"] = ["double", "float", "half"]
22 tune_params["XTYPE"] = ["double", "float", "half"]
23 tune_params["ATYPE"] = ["double", "float", "half"]
24 tune_params["VECTOR_SIZE"] = [1, 2, 3, 4]
25
26 tune_kernel("axpy", "axpy.cu", n, args, tune_params, answer=answer, observers=observers, metrics=metrics,
27             objective="fitness", strategy="genetic_algorithm", compiler_options=["-std=c++17"],
28             grid_div_x=["BLOCK_SIZE_X", "VECTOR_SIZE"])
```

Listing 2: Example of tuning script to tune the `axpy` kernel from Listing 1 using Kernel Tuner.

objective function values. Crossover uses either single-point, two-point, uniform, and disruptive uniform crossover. After crossover, invalid configurations are mutated and then repaired, if needed. Mutation replaces an individual with a random valid neighbor that differs in exactly one parameter.

PSO Particle Swarm Optimization starts from a randomly sampled population of valid configurations. The inertia, cognitive, and social coefficients are used to move the particles around according to their own best solution and the global best solution found so far.

MLS Multi-start Local Search starts from a random valid configuration and uses hill climbing as the local search method. Specifically, we use greedy strategy that immediately moves to a new configuration as soon as a better solution is found. When stuck in a local optimum, the search restarts from a new random configuration.

BO Bayesian Optimization consists of three main building blocks: initial sampling, surrogate modeling, and acquisition. Kernel Tuner's BO uses Latin Hypercube Sampling to draw an initial sample. The surrogate model, a Gaussian Process, approximates the unknown objective function and captures both the fitted data and associated uncertainty. The acquisition function proposes new valid configurations to evaluate next by automatically learning the best performing acquisition function starting from a portfolio that includes Probability of Improvement, Expected Improvement, and Lower Confidence Bound.

BH Basin Hopping is a stochastic algorithm for finding the global minimum of a function in the presence of many local minima separated by large barriers. Starting from a random valid configuration, BH iteratively perturbates the current position, then a local minimization algorithm, L-BFGS-B by default, is used to find the new local minimum, which is then accepted or rejected based on the returned value and current temperature.

FF Firefly is implemented as a variant of PSO with a slightly different method for updating particle positions. The implementation uses the same sampling and repair techniques as PSO.

SA Simulated Annealing starts from a random valid configuration and uses an annealing schedule based on a start temperature T, a minimum temperature T_{min}, and parameter α that controls the cooling rate from T to T_{min}. The annealing schedule is scaled when the user also explicitly passes a maximum number of function evaluations, to ensure the algorithm uses the full budget. At each annealing step, the current configuration is perturbed, which may or may not be accepted depending on the objective value and current temperature. The algorithm restarts if stuck.

F. Illustrative Example

Listing 2 shows an example of the Python code required to tune the `axpy` kernel from Listing 1. Lines 58-8 define the kernel's input and output data. Line 10 specifies the kernel arguments, following the same order and types as used by the kernel's signature. Here, we use the `TunablePrecision` to signal to the tuner that the data types of `a`, `x` and `y` depend on the tunable parameters `ATYPE`, `XTYPE`, and `YTYPE`. Line 12 specifies the reference solution, computed using NumPy, to be used by the `AccuracyObserver` on Line 13 when computing the error metric. Lines 15-17 specify the user-defined metrics, including performance measured in GFLOP/s and a fitness function $f(x)$ for accuracy-aware tuning. This example uses a *hard* penalty that simply returns zero if the log-error exceeds -7. Lines 17-24 specify the tunable parameters of our kernel: `BLOCK_SIZE_X` for the thread block x-dimension and `ATYPE`, `XTYPE`, and `YTYPE` specify the floating-point type to use for the variables `a`, `x`, and `y`, respectively. Line 28 finally calls the tuner, specifying the name of the kernel, the kernel code, the domain to parallelize, the argument list, tunable parameters, reference solution, the `AccuracyObserver`, user-defined metrics, the

TABLE II
BENCHMARKS USED FOR PERFORMANCE EVALUATION. PARAMETERS ARE DIVIDED INTO PRECISION (PREC.) AND OPTIMIZATION (OPT.) CATEGORIES. THE EXACT NUMBER OF VALID CONFIGURATIONS DEPENDS ON HARDWARE CAPABILITIES, NUMBERS SHOWN FOR NVIDIA A100.

Name	**Field**	**Source**	**Error Metric**	**No. of parameters**			**No. of configurations**		
				Opt.	*Prec.*	*Total*	*Opt.*	*Prec.*	*Total*
Bessel	Mathematical Physics	Kernel Float [40]	MRE	2	5	7	16	189	3,024
Convolution2D	Image Processing	BAT [44]	NRMSE	7	4	11	864	112	72,112
K-Means	Machine Learning	Rodinia [45]	MRE	3	3	6	52	64	3,328
LavaMD	Molecular Dynamics	Rodinia [45]	NMAE	3	5	8	48	336	14,896
InvK (Inverse Kinematic)	Robotics	AxBench [46]	NMAE	2	5	7	20	240	4,641
Newton-Raphson	Numerical Analysis	AxBench [46]	NRMSE	2	6	8	12	468	5,572
Coulombic	Electrical Engineering	Parboil [47]	MRE	4	4	8	144	144	20,736
MRI-Q	Medical	Parboil [47]	NRMSE	3	5	8	48	640	29,916
Black-Scholes	Finance	CUDA Samples [48]	NRMSE	2	5	7	12	576	6,383

objective for optimization, the optimization strategy to use (Genetic Algorithm in this case), compiler options, and grid divisors.

IV. BENCHMARK KERNELS

To evaluate our work, we selected GPU kernels representing a variety of workloads from different areas of scientific computing (see Table II). The kernels are as follows:

Bessel Calculates the modified Bessel function of the first kind, zeroth order. These equations appear in problems involving wave propagation and static potentials.

Convolution2D This applies a 2D convolution filter to an input image, a common task in image processing. The tunable parameters include the thread block size, the tile size, the vector width, and shared-memory usage. Additionally, the input, filter, accumulator, and output variables have tunable data types. For our benchmark, we apply a $17{\times}17$ filter to an $8192{\times}8192$ image.

K-Means The K-means algorithm, common in data analysis, partitions N data points (with D features) into K clusters. This kernel performs the most compute-intensive part: finding the nearest cluster center for each point. Our benchmark uses $N = 5 \times 10^6$, $K = 40$, and $D = 15$.

Inverse Kinematics Performs inverse kinematics for a four-jointed robotic arm, i.e., calculating the joint angles required for the arm to reach each of the N target positions. This requires several transcendental functions, including an inverse trigonometric operation.

Newton-Raphson Finds a root of a fourth-degree polynomial via Newton–Raphson. For the benchmark, we process 10^8 polynomials in parallel, performing 10 iterations each. The first 5 and last 5 iterations use different data types and thus can be performed at different precision levels.

Coulombic Computes the electric potential at discrete grid points given a set of charged particles. The positions, charges, and potentials have tunable data types. Our benchmark uses 10^4 particles and 4×10^6 grid points.

MRI-Q Reconstructs the spatial data from k-space frequency data by an inverse non-uniform Fourier transform, a compute-intensive task in MRI analysis requiring many trigonometric operations. Our benchmark uses dimensions of $64{\times}64{\times}64$ for the spatial domain and $16{\times}16{\times}16$ for the k-space domain.

Black-Scholes A mathematical model from quantitative finance for calculating option pricing. It makes heavy use of transcendental operations such as exponential and logarithm functions. The benchmark processes 2×10^8 options in parallel.

LavaMD Calculates the inter-atomic forces among particles in three-dimensional space, a common task in computational chemistry. The simulated space is divided into fixed-size cubes and each thread block sums the particle forces for one cube and its 26 neighbors. Our benchmark uses 8.6×10^5 particles divided into $15{\times}15{\times}15$ cubes.

For each kernel, we modified the GPU code from the original source to introduce several tunable parameters and configurable data types. We achieved this by integrating *Kernel Float* [40], a header-only CUDA/HIP library that simplifies working with vector and low-precision arithmetic on GPUs. Specifically, for each kernel, we introduced two kinds of tunable parameters: *optimization* and *precision* parameters.

The *optimization* parameters control how the kernel is launched and how the workload is parallelized without changing numerical accuracy. Typical examples are thread-block size, loop-unroll factor, vector width, and the use of shared memory. Note that, due to certain restrictions on these parameters, not all combinations can be evaluated.

In contrast, *precision* parameters affect both the performance of the kernel and the accuracy of the results. There are three kinds of precision parameters. First, there are parameters that define the data types for intermediate floating-point variables. These affects only computational characteristics such as the floating-point operations performed and register usage. Second, there are the data formats for the input or output buffers. These also impact memory performance such as memory bandwidth, cache utilization, and the memory footprint. Third, for kernels that rely on transcendental functions (e.g., trigonometry), Kernel Float lets us set the *execution policy*.

For the tunable data types, we provided four possible values: FP64, FP32, FP16, and BF16. Although FP16 and BF16 occupy the same amount of memory, the limited exponent range of FP16 can cause over- or underflows that BF16 avoids. The execution policy can be set to `accurate` (use the most accurate implementation available), `fast` (prefer fast-math intrinsics when available, otherwise fall back to `accurate`), or `approx` (use Kernel Float's approximate variant when available, otherwise

fall back to `fast`). The `fast` policy applies only to FP32 and `approx` to FP16/BF16.

To measure the numerical accuracy of the kernels, we implemented each kernel in Python in double precision. The outputs y_i of the GPU kernel are compared against the reference outputs y_i' of these Python implementations. We excluded configurations that resulted in invalid outputs (NaN or infinity) due to overflow. For each benchmark, we selected one of the following metrics (see Table II):

$$\text{log-MRE} = \log_{10}\left(\frac{1}{n}\sum_{i=1}^{n}\frac{|y_i - y_i'|}{|y_i'|}\right),$$

$$\text{log-NRMSE} = \log_{10}\left(\frac{\sqrt{\frac{1}{n}\sum_{i=1}^{n}|y_i - y_i'|^2}}{\frac{1}{n}\sum_{i=1}^{n}y_i'}\right),$$

$$\text{log-NMAE} = \log_{10}\left(\frac{\sum_{i=1}^{n}|y_i - y_i'|}{\sum_{i=1}^{n}|y_i'|}\right).$$

The *mean relative error* (MRE) is simply the average relative error across all values, meaning it cannot be used if the reference values are close to zero. The *normalized root mean squared error* (NRMSE) measures the normalized average squared error, thereby penalizing large outliers more heavily, but it cannot be used if the mean over the reference values is near zero. Finally, the *normalized mean absolute error* (NMAE) is the ratio of the total error magnitude to the total data magnitude. We take the *base-10 logarithm* of these metrics so that the result indicates the number of correct decimal places; for example, a log-MRE of $-k$ means that, on average, the first k decimal places are correct.

To measure the performance of low-precision configurations, we measure the speedup over the best FP64 configuration for each kernel. This baseline is the fastest configuration using FP64 types and the `accurate` execution policy.

V. Performance Evaluation

In this section, we evaluate our accuracy-aware extension to Kernel Tuner. First, we describe our experimental setup (Section V-A) and analyze the performance–accuracy trade-off of the kernels (Section V-B). Next, we introduce an accuracy-aware objective function (Section V-C) and use it to evaluate eight search strategies in Kernel Tuner (Section V-D), comparing them against each other (Section V-E) and related work (Section V-F). Finally, we demonstrate that our unified tuning approach outperforms sequential tuning (Section V-G), evaluate its effectiveness under alternative objectives (Section V-H), and apply it to a full application (Section V-I).

A. Experimental Setup

We conducted our experiments on two different GPUs: the Nvidia A100 and the AMD Instinct MI250X.

For the A100, we used the ASTRON site of DAS6 [49], the distributed supercomputer of the Netherlands. This GPU is based on Nvidia's Ampere architecture and has 40 GB of memory. Each DAS6 node is equipped with two 16-core AMD EPYC 7282 CPUs and 128 GB of memory. Software versions for our experiments were Rocky Linux 8.9, CUDA 12.2.1, Python 3.11.5, and Kernel Tuner 1.3.

For the MI250X, we used LUMI [50], a pre-exascale system located in Finland. The GPU partition of LUMI has 2,978 nodes, each with one 64-core AMD Trento CPU and four GPUs. The Instinct MI250X GPU is based on AMD's CDNA2 architecture and has 128 GB of HBM2E memory. Note that the MI250X has two compute dies and our experiments were performed on only a single die. The software environment consisted of SUSE Linux Enterprise Server 15 with ROCm 6.0.3, Python 3.9.13, and Kernel Tuner 1.3.

B. Performance-Accuracy Trade-Off

Figs. 2 and 3 show the results for the A100 and MI250X when using the brute-force search strategy, which evaluates all valid kernel configurations. Each graph plots the log-error (horizontal axis) against the speedup over FP64 (vertical axis) for all evaluated configurations. Note that these errors are not directly comparable across graphs, since their interpretations differ for each kernel. To indicate the magnitude of the numerical error, the graphs include vertical lines corresponding to the error obtained when all tunable data types are set to FP64, FP32, FP16, or BF16.

From these figures we observe that a performance gain over FP64 can be achieved by tolerating some loss in accuracy for every kernel on both GPUs. In other words, each graph illustrates the mixed-precision trade-off between numerical accuracy and performance. Some configurations are optimal in the sense that no other configuration offers both higher accuracy and better performance. These configurations are called *Pareto-optimal* and they are highlighted in the figures. The set of Pareto-optimal points is called the *Pareto front*.

The Pareto front varies per kernel, per GPU, and per error. For example, for MRI-Q on the A100 the maximum speedup obtained is $12.3\times$ at a log-error of -1.9, whereas on the MI250X the speedup reaches $9.7\times$ at a log-error of -5.9. On the other hand, for InvK on the A100 the maximum speedup is $4.4\times$ at a log-error of -3.4, while on the MI250X it reaches up to $12.3\times$ at a similar error level.

Another observation is that some of the Pareto-optimal configurations are truly mixed-precision in the sense that these configurations combine different precision levels to trade off accuracy and performance. For example, on A100, the Convolution2D, K-Means, Coulombic and Black-Scholes kernels contain Pareto-optimal configurations with accuracy between FP32 and FP16, outperforming full-FP32 configurations. We observe a similar effect on MI250X between FP64 and FP32 for the Bessel, Convolution2D, and MRI-Q kernels.

C. Objective Function

To utilize Kernel Tuner's search strategies, we define an objective function as specified in Section III-D. In this work, we consider three possible options for the penalty function:

$$\text{penalty}_{\text{hard}}(x) = 0,$$

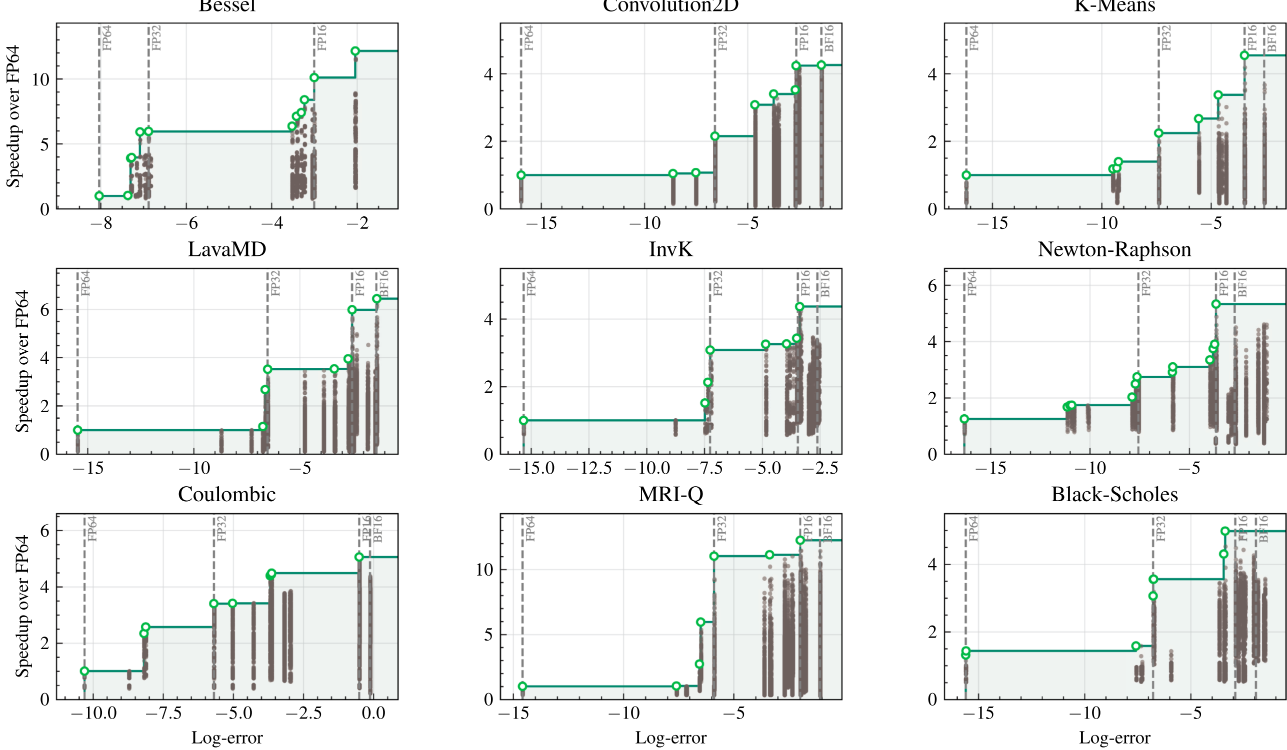

Fig. 2. Speedup versus numerical error for Nvidia A100. Each point corresponds to one evaluated configuration. The highlighted points are considered Pareto optimal. Eech vertical dashed line indicate the error obtained when all tunable data types are set to one specific type.

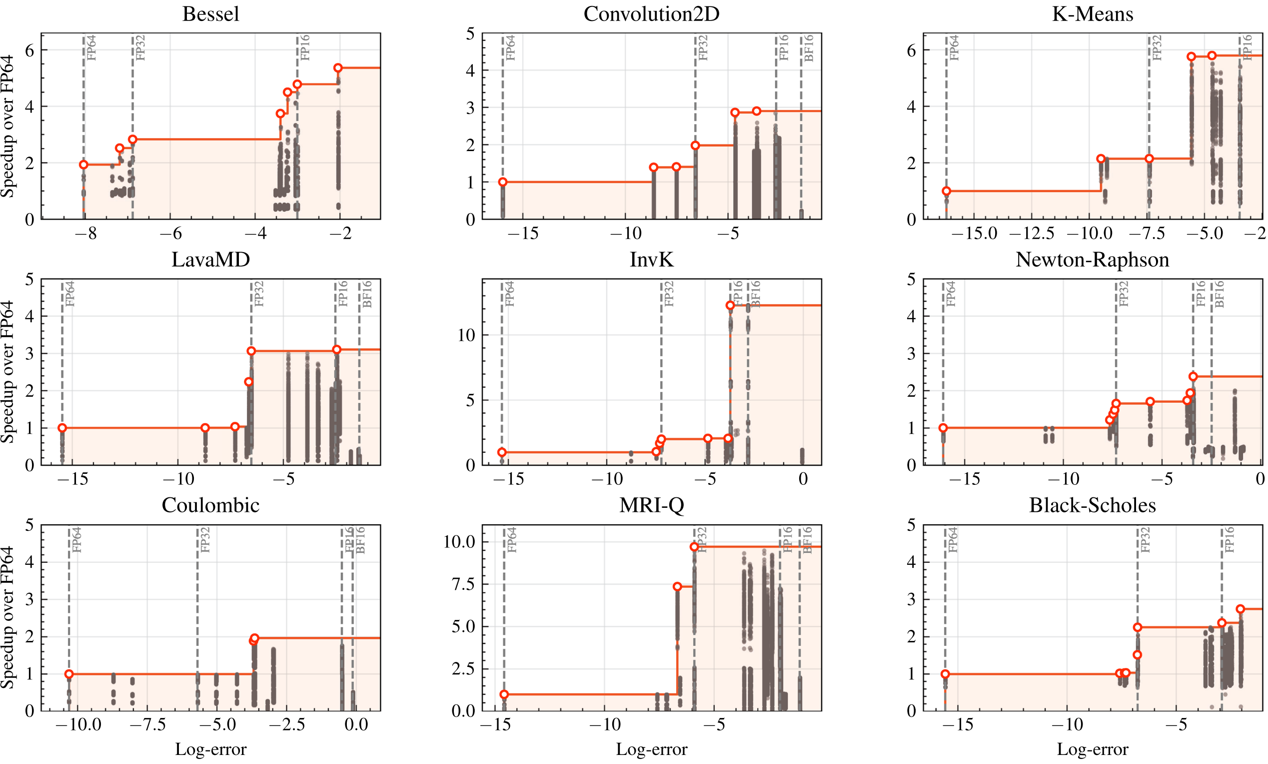

Fig. 3. Speedup versus numerical error for AMD MI250X.

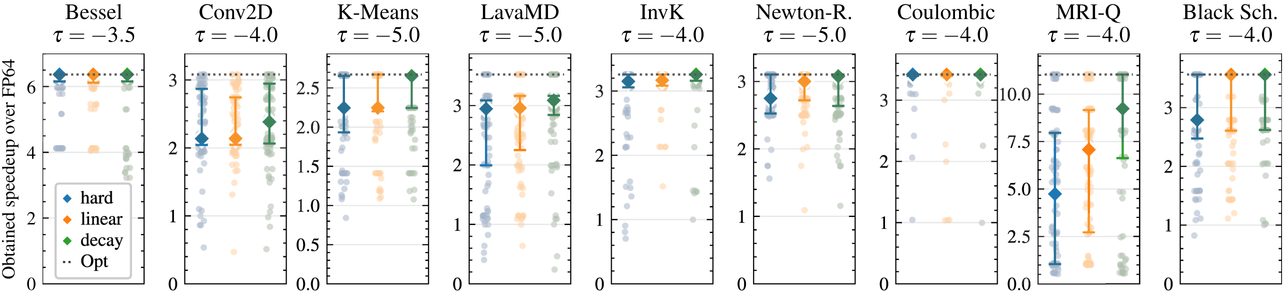

Fig. 4. Speedups of the three penalty functions for the nine kernels on Nvidia A100. Light dots show all 100 runs, the diamond indicates the median, error bars span the quartiles, and the dotted line shows the optimal speedup for the chosen threshold τ.

TABLE III
HYPERPARAMETERS VALUES FOR THE SEARCH STRATEGIES

Algorithm	Hyperparameter	Values
Random (**Rand**)	Fraction	1.0
Genetic Algorithm (**GA**)	Method Population size Mutation chance Max. iterations	single-point 26 55 90
Particle Swarm Optimization (**PSO**)	Population size Max. iterations w c_1 c_2	50 190 0.5 3.5 1.0
Multi-start Local Search (**MLS**)	Neighbor	Hamming-adjacent
Bayesian Optimization (**BO**)	Covariance kernel Covariance scale Sampling method Population size Method	matern32 1.5 LHS 20 multi-ultrafast
Basin Hopping (**BH**)	Method Temperature	L-BFGS-B 1.0
Firefly (**FF**)	Population size Max. iterations B_0 γ α	20 100 1.0 1.0 0.2
Simulated Annealing (**SA**)	T T_{min} α Max. iterations	0.1 0.001 0.9975 1

$$\text{penalty}_{\text{linear}}(x) = \alpha(\tau - \text{log-error}_x),$$

$$\text{penalty}_{\text{decay}}(x) = \text{speedup}_x \cdot \exp(\beta(\tau - \text{log-error}_x)).$$

The *hard* penalty always returns zero, which is straightforward but risks trapping the search strategy as there is no gradient to escape infeasible regions. The *linear* penalty decreases proportionally with the error, encouraging the strategy to move toward configurations with lower error while disregarding the performance. The *decay* penalty multiplies the performance by the exponential of the error, pushing the strategy toward a region of lower error and higher performance.

To evaluate these penalty functions, we used Kernel Tuner's genetic algorithm (GA) with default settings (See Table III) on the Nvidia A100. For each penalty function and each kernel, we performed 100 runs, with 250 evaluations per run, using α=β=1. Because the strategies are stochastic, each run finds a slightly different result depending on the random seed. Fig. 4 shows the median speedups obtained over the runs relative to the optimum.

The graphs indicate that, for all kernels, the hard penalty performs either worse or no better than the others. The decay penalty outperforms the linear penalty for six out of the nine kernels and is on par with linear for the other three kernels. From this, we can conclude that the decay penalty performs best, closely followed by the linear penalty, and the hard penalty performs the worst. We will use the decay penalty in the remainder of this work.

D. Search Strategy Evaluation

Next, we will evaluate the effectiveness of eight Kernel Tuner search strategies (See Section III-E). In Kernel Tuner, the optimization budget can be set as either a maximum number of unique function evaluations, a wall-clock time, or both. Kernel Tuner uses a memoization scheme to avoid evaluations of configurations that have already been compiled and benchmarked. Furthermore, a Constraint Satisfaction Problem (CSP) solver is used to construct the search space of all valid configurations at the start of the tuning process [51]. We use the default hyperparameters in Kernel Tuner for each strategy, as listed in Table III. See [52] for a description of how the hyperparameters were selected.

As the error distributions differ across kernels (see Figs. 2 and 3), for each kernel, we manually selected thresholds corresponding to three precision levels (see Table IV):

High precision Between full FP64 and full FP32.
Medium precision Between full FP32 and full FP16.
Low precision Lower than full FP16.

For each combination of the eight strategies, nine kernels, two GPUs, and three thresholds, we performed 100 runs using Kernel Tuner. In every run, we set the tuning budget to 250 configurations and a wall-clock time limit of 15 minutes, and selected the fastest configuration that met the error threshold. Because the numerical error of a configuration is known only

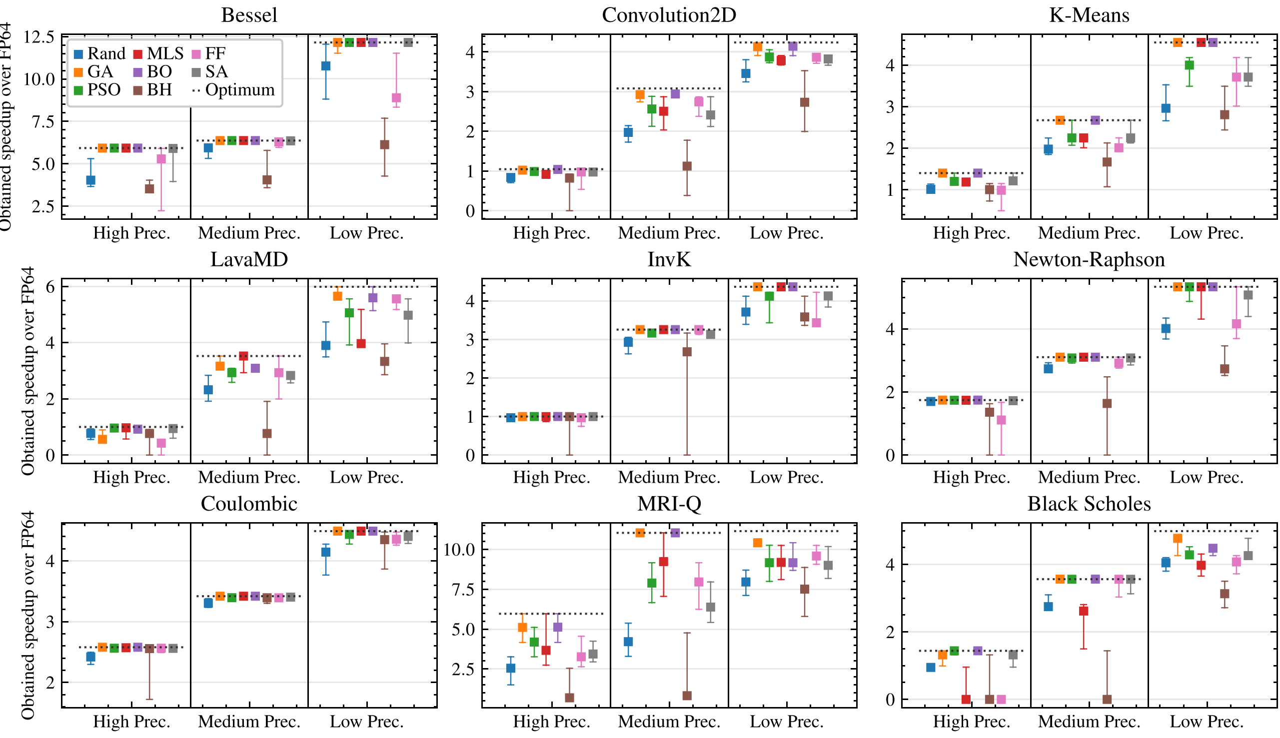

Fig. 5. Each graph marks the *median* performance of the best configurations found by every search strategy on a specific kernel at a given accuracy threshold. The error bars shows the quartiles and the dotted line indicates the global optimum.

TABLE IV
ACCURACY LEVELS OF DIFFERENT KERNELS. EACH ENTRY LISTS THE THRESHOLD AND THE RELATIVE SEARCH SPACE SIZE (%).

Kernel	High		Med.		Low	
Bessel	−7	(7%)	−3.5	(25%)	−2	(100%)
Convolution2D	−8	(2%)	−4	(16%)	−2	(73%)
K-Means	−8	(6%)	−5	(19%)	−3	(75%)
LavaMD	−7	(4%)	−5	(12%)	−2	(62%)
InvK	−8	(2%)	−4	(10%)	−3	(46%)
Newton-Raphson	−8	(4%)	−5	(23%)	−2	(72%)
Coulombic	−7	(6%)	−4	(22%)	−2	(50%)
MRI-Q	−6	(3%)	−4	(6%)	−2	(39%)
Black Scholes	−8	(1%)	−4	(14%)	−2	(73%)

after benchmarking, any configuration whose error exceeded the threshold was *rejected*. These rejected configurations still counted toward the evaluation limit.

Fig. 5 shows results for the A100, reporting the median performance of the best configuration found across the different runs, with error bars indicating Q1 and Q3. Each graph is divided into three regions corresponding to the high, medium, and low precision thresholds. The graphs show substantial variation across kernels: for some kernels, the search strategies typically locate the optimum, whereas for others they do not. For example, for Bessel, the search strategies are typically able to find the global optimum. This is due to the small search space (approximately 3,000 configurations), such that evaluating 250 of them covers about 8% of the total.

However, larger search spaces do not always behave similarly. For instance, for Coulombic, the strategies usually find a near-optimal configuration, even though the search space is relatively large (over 20,000 configurations). Here, we also see that random sampling performs well, suggesting that the search space, while large, is “easy” as many configurations deliver near-optimal performance. The opposite occurs for MRI-Q, where the search space is also large (almost 30,000 configurations), yet most strategies struggle to find the optimum.

We also observe variation across accuracy levels within the same kernel. For InvK, the strategies perform well at high and medium precision, but they struggle at low precision (except for GA, MLS and BO). This suggests that optimizing is easier at higher precision levels than at lower ones. On the other hand, for K-Means three strategies reliably find the optimum at low precision, but only two on medium or high precision. For MRI-Q, the optimum is only consistently returned by GA and BO at medium precision.

Additionally, several strategies at the high-precision threshold show a median speedup of zero, meaning they failed to find *any* configuration that satisfied the accuracy constraint. As shown in Table IV, for high precision the search space is highly sparse: there are very few configurations that have an accuracy above the error threshold. At medium and low precision the space is denser, allowing the strategies to find near-optimal configurations more reliably.

As the number of configurations evaluated by each search strategy is limited to 250, we must also consider the wall-clock time spent by the tuner. Fig. 6 shows the average time spent by

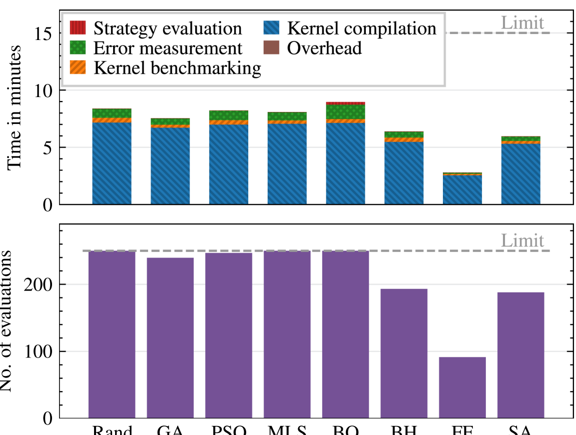

Fig. 6. Average wall-clock time (top) and number of evaluations (bottom) across the different runs from Fig. 5.

Kernel Tuner across all runs corresponding to Fig. 5. The top graph shows that the average tuning time is around 8 minutes for most strategies, which corresponds to roughly 2 seconds per configuration evaluation. BH, SA, and FF spend less time on average because, as the bottom graph shows, they typically do not exhaust the evaluation budget. The total tuning time is divided into five components, revealing that compilation is the most expensive part of the tuning process. For example, for BO, approximately 80% of the time is spent on kernel compilation, 13% on error measurement, 3.8% on kernel benchmarking, and only 2.6% on the Bayesian optimization method itself.

E. Comparison Across Search Strategies

In the previous section, we evaluated eight search strategies. This showed that a strategy might perform great in one case, but poorly in another. To quantify this behavior, we can calculate – for each pair of two strategies – the *probability* that a single run of the first strategy yields a faster configuration than a single run of the second, depending on the random seed and breaking ties at random. This measures the *relative* performance of each search strategy: 50% indicate they perform equally well and higher/lower values indicate higher/lower performance.

Fig. 8 shows an example of these probabilities for the *Bessel* kernel on the Nvidia A100 at low and medium precision. We see, for example, that BO performs exceptionally well as the probability to outperform random sampling is 96% at high precision and 97% at medium precision. Firefly, on the other hand, performs poorly at high precision (48% to outperform random sampling), but decently at medium precision (70%). Basin-Hopping always performs poorly, with the probability of outperforming any other strategy being at most 44% . GA outperforms PSO at medium precision, but the reverse is the case for high precision.

To simplify the analysis, for the remainder of this section, we treat random search as our neutral baseline and focus solely on the probability that a strategy outperforms random sampling. The larger this probability, the stronger the strategy. Fig. 7 shows the probability of outperforming random sampling for each of the seven other strategies.

Overall, Bayesian Optimization (BO) and Genetic Algorithm (GA) perform exceptionally well. Particle Swarm Optimization (PSO), Multistart Local Search (MLS), and Simulated Annealing (SA) perform on par and have a good chance of outperforming random sampling. Firefly (FF) is only slightly better than random sampling, and Basin Hopping (BH) performs much worse. Interestingly, when we look at how search spaces differ across GPUs, we do not see substantial variation in the performance of the search strategies.

On the other hand, tuning kernels with high precision, between FP64 and FP32, shows that BO consistently outperforms all other strategies, performing best at high precision and slightly worse at medium and low precision. In contrast, MLS and GA perform increasingly better for lower precisions. The reason is that the high precision tuning cases leave large gaps in the search space, making those strategies less effective, while the BO strategy in Kernel Tuner has been designed specifically to efficiently deal with highly-fragmented search spaces [33]. Independent of the precision level, all strategies except BH, consistently outperform random sampling.

When comparing search strategy performance across different kernels, we see some differences. For example, MLS performs the best only on InvK and Coulombic, while GA wins for K-Means and Newton-Raphson. For the remaining kernels, BO consistently outperform the others.

A notable outlier where the strategies struggle is LavaMD at high precision. This occurs because, for example on the A100, the maximum achievable speedup at the high threshold ($\tau = -7$) is only $1.15\times$, whereas a much larger $3.5\times$ speedup is obtainable with a log-error of -6.51. This exposes a weakness in the decay penalty: it is not strict enough to discourage near-threshold configurations. This issue can be resolved by increasing β. For example, for LavaMD at high precision ($\tau{=}{-}7$), raising β to 2.3 resolves this as then $3.5 \cdot \exp\big(\beta(\tau - (-6.51)\big) \approx 1.13$, which is lower than 1.15.

F. Comparison of Search Strategies With Related Work

In the previous section, we compared the performance of mixed-precision tuning for seven strategies from Kernel Tuner against random sampling. This comparison showed that, on average, BO (Bayesian Optimization) and GA (Genetic Algorithm) performed best. However, the software architecture of Kernel Tuner is flexible enough to integrate search strategies from other auto-tuning frameworks [53] as well. This flexibility allows us to delegate the search to alternative implementations while still using Kernel Tuner's features, such as code compilation, error metric calculation, and accuracy-aware objective evaluation. This enables us to compare our approach against the search strategies of other auto-tuning frameworks, including those that currently lack support for mixed-precision and accuracy-aware tuning.

For this comparison, we considered two general-purpose auto-tuning frameworks: PyATF [28] and ytopt [30] (which relies on a fork of *scikit-optimize*[4]). For PyATF, we evaluated

[4]https://github.com/ytopt-team/scikit-optimize

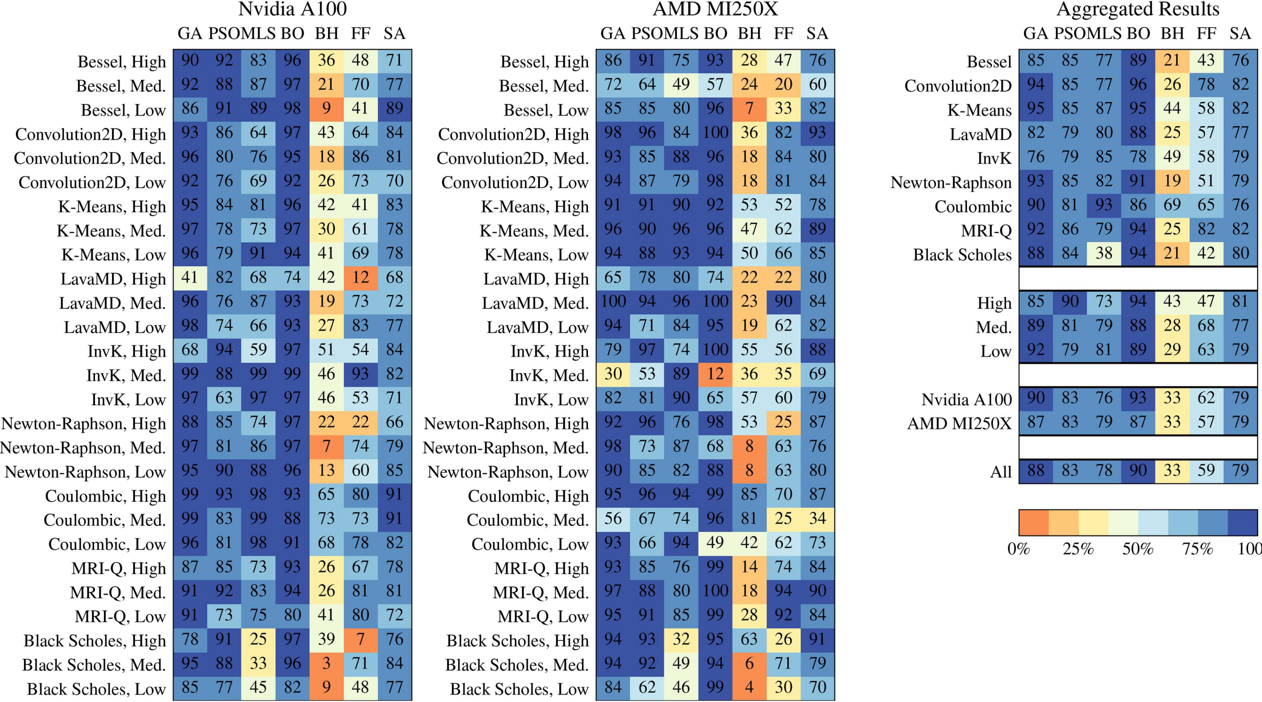

Fig. 7. Probability (%) that each strategy (columns) outperforms random sampling across multiple scenarios (rows). The right-most table reports averages calculated per kernel, per precision level, per GPU, and the overall.

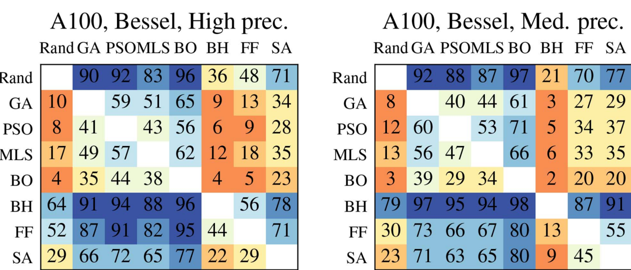

Fig. 8. Pairwise probability (%) that one strategy (column) outperform another strategy (row) for two precision levels.

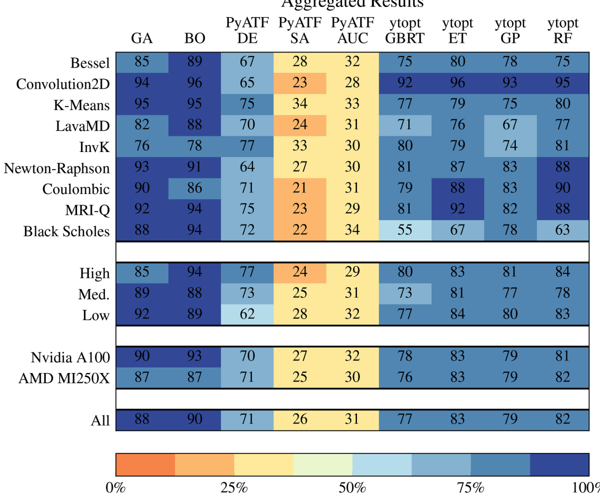

Fig. 9. Probability (%) that each strategy (columns) outperforms random sampling. Results are aggregated per kernel, per precision level, per GPU, and overall.

three strategies: *Differential Evolution* (DE), *Simulated Annealing* (SA), and *AUC Bandit* (AUC). For ytopt, we used their BO implementation with four different surrogate models: *Gradient-Boosted Trees* (GBRT), *Extra Trees* (ET), *Gaussian Processes* (GP), and *Random Forest* (RF). As before, we performed 100 runs, with each run limited to 250 evaluations and a time limit of 15 minutes.

Fig. 9 shows the averaged results for Kernel Tuner (BO and GA), PyATF, and ytopt. The results indicate that, on average, GA and BO outperform the algorithms from PyATF and ytopt, with average probabilities of 88% and 90% , respectively. Within PyATF, DE performs best with an average probability of 71% , while SA and AUC appear to perform worse than random sampling (26% and 31%).

For ytopt, all four algorithms perform reasonably well, and in some cases their performance is similar to that of Kernel Tuner's BO. Among them, ET and RF perform best, with average probabilities of 83% and 82% . However, it should be noted that ytopt is slow as the average tuning time is 11.6 minutes for GBRT, 11.5 for ET, 14.9 for GP, and 11.6 for RF, while Kernel Tuner's BO takes only 8.7 minutes.

G. Need for Unified Tuning

As described in Section IV, there are two classes of tunable parameters: (*a*) *optimization parameters*, which influence

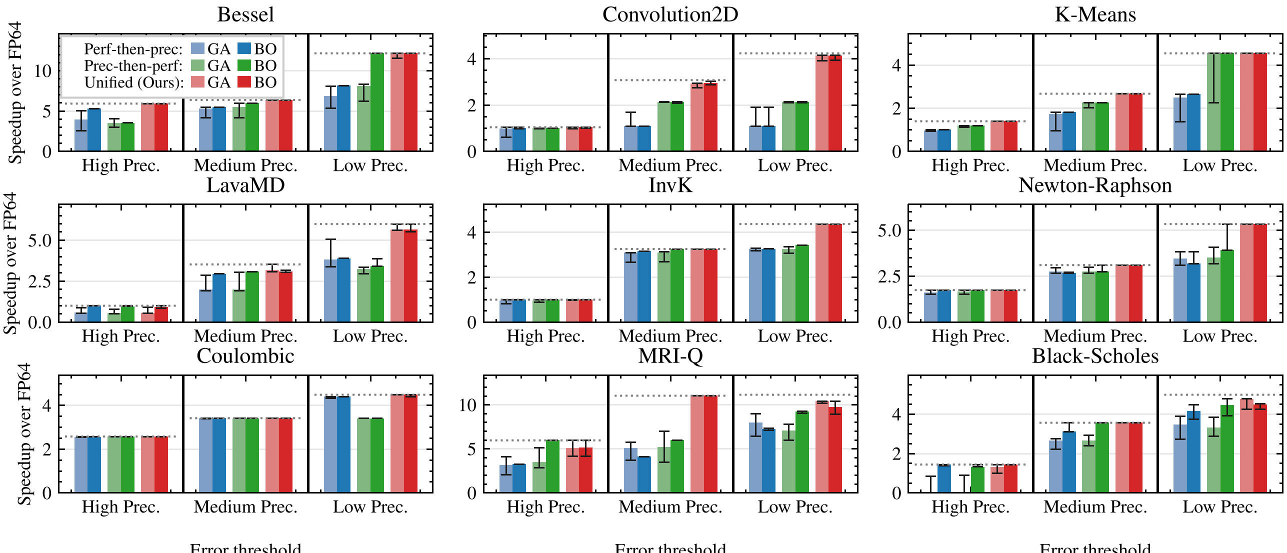

Fig. 10. Median speedup obtained for the two alternative methods described in Section V-G compared to our unified method, using BO or GA. Results from Nvidia A100 for different kernels on three precision levels. Error bars show quartiles.

performance without affecting numerical accuracy, and (*b*) *precision parameters*, which affect both accuracy and performance. As discussed in Section II, most existing auto-tuners can either tune optimization parameters without considering numerical error (Section II-A) or tune precision parameters without considering performance (Section II-B). While one could hypothetically select data types using a precision-tuning tool and then choose the optimization parameters with a generic auto-tuning framework (or vice versa), as we show in this section, such a sequential approach does not always result in optimal performance.

This is because there are often subtle interactions between these two classes of parameters. For example, reducing a data type from FP64 to FP32 lowers the register usage per thread, freeing resources that allow a larger thread-block size and potentially increasing overall performance. Consequently, optimizing for performance and optimizing for accuracy are not independent problems, but should be addressed *simultaneously*. The strength of our **unified approach** lies in its ability to capture these interactions by jointly tuning optimization and precision parameters. However, it is unclear whether the search strategies will be able to handle the much larger unified search space, and whether a unified approach can in practice outperform approaches that optimize both problems separately. We compare the following three approaches:

Performance-then-Precision: Fix the precision parameters to FP64 and tune only the optimization parameters to find the configuration with the lowest run time. This is effectively the approach of the performance tuners listed in Section II-A, which cannot measure or optimize numerical accuracy. Next, fix these optimization parameters and tune the precision parameters to identify the fastest configuration whose numerical error remains below a specified threshold. This is effectively the approach taken by the precision tuners listed in Section II-B, which assume reducing precision improves performance.

Precision-then-Performance: The reverse of the above: first precision parameters and then optimization parameters.

Unified: Jointly tune both optimization and precision parameters as a single tuning process.

For the unified strategy, we use a budget of 250 configuration evaluations. For the two sequential strategies, this budget must be split between their two stages, determining how much *effort* is spent on optimization versus precision tuning. For this work, we divide the budget in proportion to the number of parameters of each type. For example, for Convolution2D, the budget split is 63.6% for optimization parameters (7 of 11) and 36.4% for precision parameters (4 of 11). While different budget splits could affect the results, we expect the impact to be small because the isolated search spaces are relatively limited in size (see Table II), and most reasonable splits should explore a large fraction of these spaces.

Fig. 10 presents the results of these approaches using GA and BO for the accuracy levels from Table IV on the Nvidia A100. Results for the MI250X are similar. Each bar shows the median outcome over 100 runs and the dotted lines indicate the optimal performance for that precision level. The results demonstrate that in some cases they may come close, whereas in others they largely fall short.

We can see, for example, that unified tuning always reaches (near) optimal results. On average across all kernels and precision levels, unified tuning converges on 96% of the global optimum speedup for BO and 94% for GA. This means both strategies perform well and BO slightly outperforms GA, as observed previously in Section V-E. Moreover, Fig. 10 demonstrate that our unified approach always outperforms or is on par with both of the two sequential methods.

Comparing the sequential strategies, we see that performance-then-precision performs slightly better than precision-then-performance. On average, performance-then-precision with BO achieves only 77% of the optimal speedup, while

precision-then-performance reaches 86% . This shows that tuning the precision and optimization parameters in mixed-precision GPU kernels cannot be treated as separate optimization problems. This demonstrates the need for a unified approach, such as including precision-tuning in a generic auto-tuning framework, as we did using Kernel Tuner. In terms of stability, we can see in Fig. 10 that the error bars are generally much shorter for the unified approach compared to both isolated approaches. For BO, the standard deviation of the fraction of optimal speedup is only 9.1% under unified tuning, compared to 21.4% for performance-then-precision and 16.0% for precision-then-performance. This indicates that unified tuning not only produces high-quality results but also produces these high-quality results more reliably.

Finally, we note that this is a remarkable result. The unified search space is the product of the two isolated search spaces, and is therefore orders of magnitude larger, as shown in Table II. Yet, the optimization algorithms are able to converge to higher-quality solutions, and do so more reliably, than when the two problem spaces are optimized in isolation. While our results indicate that this joint optimization is beneficial for the problem sizes studied here, it is well known that the performance of Bayesian optimization based on Gaussian processes may degrades in very high-dimensional spaces [54]. For larger problem sizes, one might therefore consider mitigation strategies for Bayesian optimization, such as dimensionality reduction [55], sparse Gaussian processes [56], trust-region optimization [57], or hybrid approaches that combine Bayesian optimization with genetic algorithms [58].

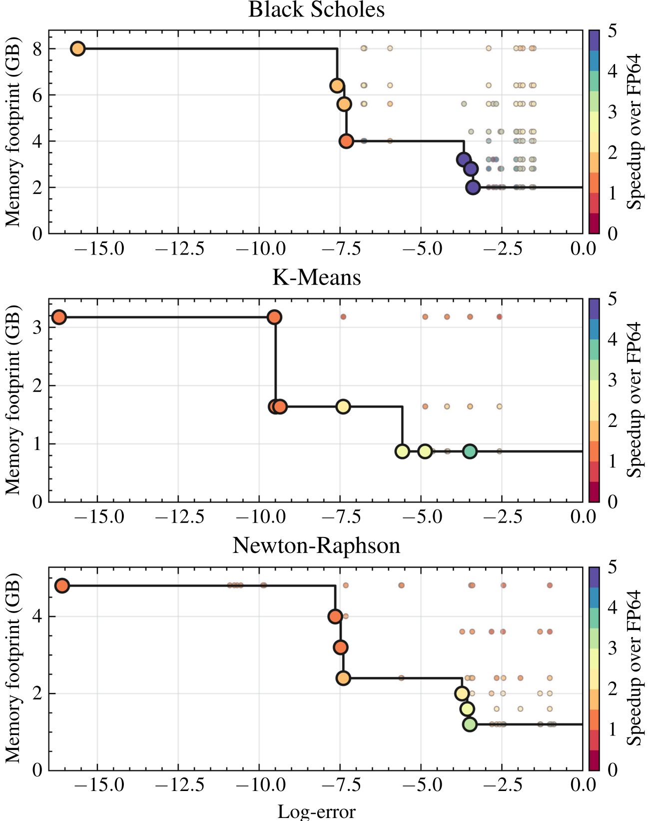

Fig. 11. Memory footprint versus error for Nvidia A100. Points are configurations; highlighted points are Pareto-optimal.

H. Alternative Objective Functions

Up to this point, our objective has been to find the configuration of parameters that maximizes the *speedup* of the kernel while keeping the accuracy above a chosen threshold. However, Kernel Tuner is flexible enough that we can also measure other metrics and optimize for user-defined objectives.

For instance, in some scenarios, rather than minimizing run time, we might want to minimize the *memory footprint* (i.e., the total memory required to store all input and output buffers) while still satisfying an accuracy constraint. With Kernel Tuner we can define user-defined objective that computes the memory footprint of each configuration. Fig. 11 plots memory footprint versus accuracy for the A100 for three kernels from Fig. 2. The maximum obtainable memory footprint reduction from FP64 to FP16 is $4\times$. The figure shows that gradually lowering the accuracy requirement also lowers the memory footprint. For example, for K-Means, imposing a maximum log-error of -5, we obtain a memory footprint of 0.87 GB (a $3.6\times$ reduction) and $2.9\times$ speedup over FP64.

Alternatively, we can use Kernel Tuner to measure the *energy consumption* of each configuration using `PowerSensor` [59] or NVML. This enables us, for example, to search for the most accurate configuration whose total energy usage stays below a specified budget (in joules), a common use case for battery-powered embedded devices. Fig. 12 shows how gradually lowering the accuracy requirement leads to improved energy efficiency on the A100 for three kernels from Fig. 2. For example, for Black-Scholes, imposing an energy budget of 1.5 J^{-1}, we achieve a log-error of -3.68, a $4.96\times$ speedup over FP64, and an energy usage of 0.64 J (a $2.7\times$ decrease over FP64). The highest energy reduction in Fig. 12 is $3.8\times$ for K-Means at a log-error of -3.49.

I. Example of Full Application

So far, we have shown how our method can be used to tune GPU kernels in *isolation*. However, real-world applications typically consist of *pipelines* of multiple GPU kernels that execute in sequence. This affects the accuracy, since the error measured for one individual kernel does not necessarily reflect the accuracy of the entire pipeline.

To evaluate our method on a more complex workflow, we apply it to *Bregman block average co-clustering with I-divergence* (BBAC-I) [60]. Co-clustering is an iterative algorithm that groups the rows and columns of an input matrix into clusters, where these dimensions correspond to space and time. This algorithm was used, for example, to study the impact of climate change based on the onset of spring in Europe [61].

The original BBAC-I application is implemented in MATLAB. We reimplemented the algorithm in double precision using CUDA. The computation time is dominated by three kernels: (1) collecting cluster statistics, (2) calculating the distances between the temporal dimension and cluster centers, and (3) updating

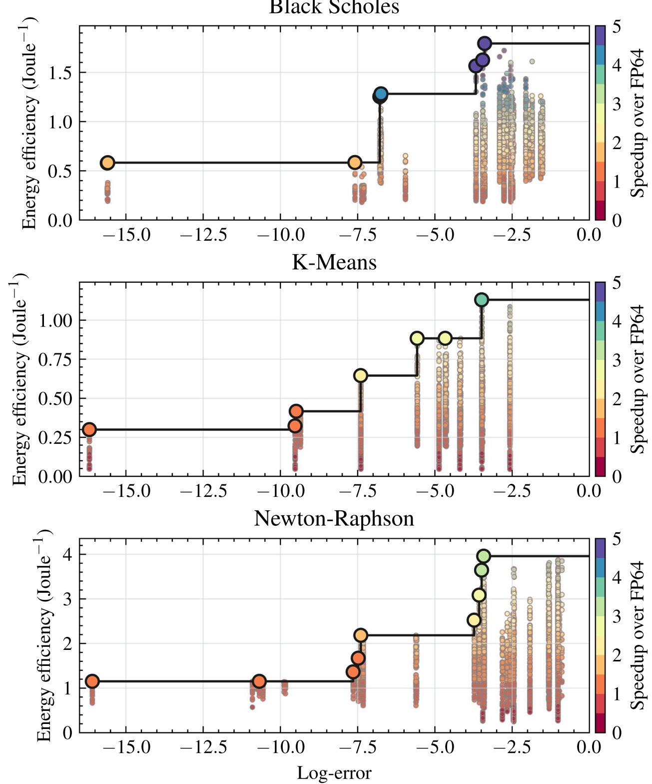

Fig. 12. Energy efficiency versus error for Nvidia A100. Points are configurations; highlighted points are Pareto-optimal.

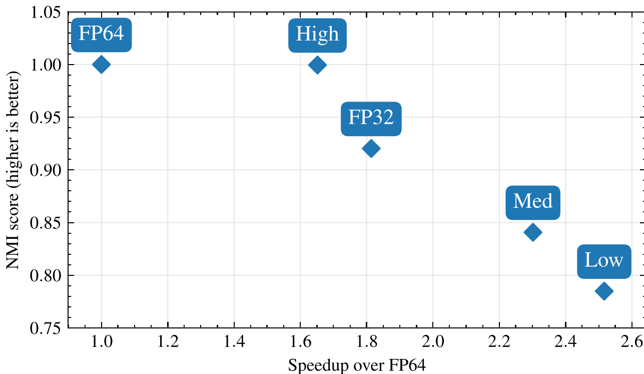

Fig. 13. Speedup versus accuracy trade-off for the five versions of the mixed-precision BBAC-I application.

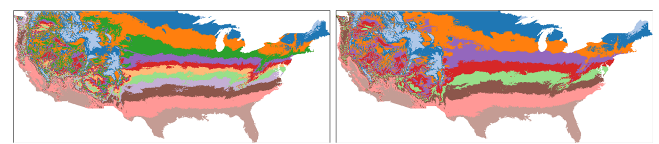

Fig. 14. Output of BBAC-I application for FP64 (left) and low precision (right). Colors represent spatial clusters.

the spatial dimension by finding the nearest cluster center. This application is well suited for lower precision, as correctness depends only on knowing which cluster center is the *closest*, not the exact distance.

We tuned each kernel using Bayesian optimization, with a maximum of 250 evaluations. When tuning each kernel individually, we used the decay penalty function and the *NMRSE* on the kernel output as the error metric. As the input matrix is shared between kernels, its data type must be fixed a priori, while the remaining data types (intermediate buffers) can be set freely. We tuned the kernels for five scenarios:

FP64: All data types set to FP64.
FP32: All data types set to FP32.
High: Error threshold $\tau = -8$, input matrix in FP32.
Med.: Error threshold $\tau = -4$, input matrix in FP16.
Low: Error threshold $\tau = -2$, input matrix in FP16.

To evaluate the full application, we performed 25 iterations of the co-clustering algorithm on an Nvidia A100 for a dataset containing $40 \times 10{,}110{,}912$ entries (3.2GB), indicating the onset of spring over 40 years in the USA. The application in FP64 takes 814 ms, excluding the time for reading the input files and writing the output file. Performance is presented as the speedup of the GPU kernel time over full-FP64 baseline, whose kernel time is 686 ms. Accuracy was defined as the agreement with the clustering produced by the full-FP64 GPU version, measured using the *normalized mutual information* (NMI) score. A score of 1.0 indicates identical clustering and 0.0 indicates no similarity. Note that we deliberately use a different metric than NMRSE for the full pipeline, as end users are concerned with the quality of the final result rather than the numerical accuracy of individual kernels.

Fig. 13 shows the results of this experiment for the A100. FP64 achieves a speedup and NMI score of 1.0, as this is our baseline. FP32 is approximately 81% faster, but the quality is lower, with an NMI score of 0.92. Interestingly, the *high*-precision version is only slightly slower than FP32 (65% speedup) and achieves an almost perfect NMI score (0.99). This demonstrates how mixed precision can yield significant speedups while barely affecting accuracy. The low-precision version is more than twice as fast (2.51$\times$ speedup), but its NMI score drops to 0.78. However, this does not necessarily indicate that the resulting clusters are meaningless, as their overall clustering pattern remains similar (See Fig. 14).

VI. Conclusion

In this paper, we introduced an accuracy-aware extension to *Kernel Tuner* for systematically exploring both performance- and precision-related tuning parameters in GPU kernels. By integrating accuracy-affecting parameters alongside traditional code-optimization parameters, our approach enables developers to make informed trade-offs that balance computational throughput against numerical error. We evaluated our approach across multiple benchmark kernels on modern Nvidia and AMD GPUs, demonstrating speedups of up to 5.1$\times$ on Nvidia hardware and 12.3$\times$ on AMD hardware compared to the double-precision baselines.

The comparison of the performance of different search strategies for tuning both precision and performance parameters

revealed that Bayesian Optimization (BO) and Genetic Algorithm (GA) have the best overall performance and show the most consistent performance across different tuning scenarios. Importantly, we further showed that jointly tuning precision with other performance-critical parameters is essential for uncovering optimal configurations as seeing them as isolated problems often yields suboptimal performance.

In addressing our central research question, our results demonstrate that the unified tuning approach, using either BO or GA, consistently outperforms isolated optimization of optimization and precision parameters despite the increase in search space size and dimensionality. Furthermore, we demonstrated that the strategies from Kernel Tuner outperform the optimization algorithms used in PyATF and ytopt.

By integrating our work as extensions to the Kernel Tuner framework, we enable the use of a wide range of optimization algorithms to optimize a diverse set of objectives along with user-controlled accuracy constraints. For example, we showed that for a K-Means kernel our approach achieves a $3.6\times$ reduction in memory footprint along with a $2.9\times$ speedup over FP64 at a log-error of -4.87, or a $3.8\times$ improvement in energy efficiency along with a $3.7\times$ speedup over FP64 at a log-error of -3.49. Finally, we have also demonstrated the application of our method on a full pipeline, BBAC-I co-clustering, and showed that mixed-precision tuning allows to trade-off accuracy for improved computational performance.

Looking ahead, several directions remain open for further research. The multi-objective nature of accuracy, memory footprint, performance, and energy suggests investigating optimization algorithms specifically for expanded design space, including advanced search strategies and machine learning-based approaches that reduce tuning times in multi-objective settings. Transfer learning techniques are also promising, where insights gained from tuning one kernel could be reused for tuning similar kernels or on different hardware. By providing a unified approach to precision and performance tuning, our extensions to Kernel Tuner lay the groundwork for a new generation of mixed-precision GPU applications that can experience significant speedups by exploiting the support for low-precision floating-point formats in modern GPUs, while preserving sufficient numerical accuracy and simultaneously lowering energy consumption and memory footprint.

Acknowledgments

Access to LUMI was granted through SURF (EINF-11263). The manuscript spelling and minor grammatical corrections were assisted by Grammarly and OpenAI ChatGPT (GPT-o3).

References

[1] Y. LeCun et al., “Deep learning,” *Nature*, vol. 521, pp. 436–444, 2015.
[2] S. Heldens et al., “The landscape of exascale research: A data-driven literature analysis,” *ACM Comput. Surv.*, vol. 53, pp. 1–43, 2020.
[3] M. Courbariaux et al., “Training deep neural networks with low precision multiplications,” 2014, *arXiv:1412.7024*.
[4] S. Gupta et al., “Deep learning with limited numerical precision,” in *Proc. Int. Conf. Mach. Learn.*, 2015, pp. 1737–1746.
[5] N. Wang et al., “Training deep neural networks with 8-bit floating point numbers,” in *Proc. Adv. Neural Inf. Process. Syst.*, 2018, pp. 7686–7695.
[6] N. Burgess, J. Milanovic, N. Stephens, K. Monachopoulos, and D. Mansell, “Bfloat16 processing for neural networks,” in *Proc. Symp. Comput. Arithmetic*, 2019, pp. 88–91.
[7] Nvidia, “Nvidia Tesla V100 whitepaper,” 2017. [Online]. Available: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf
[8] AMD, “Introducing CDNA 4 architecture,” 2025. [Online]. Available: https://www.amd.com/content/dam/amd/en/documents/instinct-tech-docs/white-papers/amd-cdna-4-architecture-whitepaper.pdf
[9] P. Hijma et al., “Optimization techniques for GPU programming,” *ACM Comput. Surv.*, vol. 55, pp. 1–81, 2023.
[10] B. van Werkhoven, “Kernel tuner: A search-optimizing GPU code auto-tuner,” *Future Gener. Comput. Syst.*, vol. 90, pp. 347–358, 2019.
[11] P. Balaprakash et al., “Autotuning in high-performance computing applications,” *Proc. IEEE*, vol. 106, no. 11, pp. 2068–2083, Nov. 2018.
[12] B. van Werkhoven et al., “Lessons learned in a decade of research software engineering GPU applications,” in *Proc. Int. Conf. Comput. Sci.*, 2020, pp. 399–412.
[13] J. Ansel et al., “OpenTuner: An extensible framework for program autotuning,” in *Proc. Int. Conf. Parallel Architectures Compilation Techn.*, 2014, pp. 303–316.
[14] C. Nugteren and V. Codreanu, “CLTune: A generic auto-tuner for OpenCL kernels,” in *Proc. IEEE 9th Int. Symp. Embedded Multicore/Many-Core Syst.-on-Chip*, 2015, pp. 195–202.
[15] A. Rasch and S. Gorlatch, “ATF: A generic directive-based auto-tuning framework,” *Concurrency Computation Pract. Exp.*, vol. 31, 2018, Art. no. e4423.
[16] M. Frigo and S. G. Johnson, “FFTW: An adaptive software architecture for the FFT,” in *Proc. IEEE Int. Conf. Acoust. Speech Signal Process.*, 1998, pp. 1381–1384.
[17] R. C. Whaley et al., “Automated empirical optimizations of software and the ATLAS project,” *Parallel Comput.*, vol. 27, pp. 3–35, 2001.
[18] R. Schoonhoven, B. Veenboer, B. Van Werkhoven, and K. J. Batenburg, “Going green: Optimizing GPUs for energy efficiency through model-steered auto-tuning,” in *Proc. IEEE/ACM Int. Workshop Perform. Model. Benchmarking Simul. High Perform. Comput. Syst.*, 2022, pp. 48–59.
[19] F. Petrovič and J. Filipovič, “Kernel Tuning Toolkit,” *SoftwareX*, vol. 22, 2023, Art. no. 101385.
[20] R. Nobre et al., “Aspect-driven mixed-precision tuning targeting GPUs,” in *Proc. 9th 7th Workshop Parallel Program. RunTime Manage. Techn. Manycore Architectures Des. Tools Architectures Multicore Embedded Comput. Platforms*, 2018, pp. 26–31.
[21] P. V. Kotipalli et al., “AMPT-GA: Automatic mixed precision floating point tuning for GPU applications,” in *Proc. Int. Conf. Supercomput.*, 2019, pp. 160–170.
[22] I. Laguna et al., “GPUMixer: Performance-driven floating-point tuning for GPU scientific applications,” in *Proc. Int. Conf. High Perform. Comput.*, 2019, pp. 227–246.
[23] R. Gu and M. Becchi, “GPU-FPtuner: Mixed-precision auto-tuning for floating-point applications on GPU,” in *Proc. 27th Int. Conf. High Perform. Comput., Data, Analytics*, 2020, pp. 294–304.
[24] H. Brunie, C. Iancu, K. Z. Ibrahim, P. Brisk, and B. Cook, “Tuning floating-point precision using dynamic program information and temporal locality,” in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2020, pp. 1–14.
[25] N. Ho et al., “GRAM: A framework for dynamically mixing precisions in GPU applications,” *ACM Trans. Architecture Code Optim.*, vol. 18, pp. 1–24, 2021.
[26] Z. Mo et al., “moTuner: A compiler-based auto-tuning approach for mixed-precision operators,” in *Proc. 19th ACM Int. Conf. Comput. Front.*, 2022, pp. 94–102.
[27] A. Rasch et al., “Efficient auto-tuning of parallel programs with interdependent tuning parameters via auto-tuning framework (ATF),” *ACM Trans. Architecture Code Optim.*, vol. 18, pp. 1–26, 2021.
[28] R. Schulze et al., “pyATF: Constraint-based auto-tuning in python,” in *Proc. 34th ACM Int. Conf. Compiler Construction*, 2025, pp. 35–47.
[29] Y. Liu et al., “GPTune: Multitask learning for autotuning exascale applications,” in *Proc. 26th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2021, pp. 234–246.
[30] X. Wu et al., “Autotuning polybench benchmarks with LLVM Clang/Polly loop optimization pragmas using Bayesian optimization,” *Concurrency Computation Pract. Experience*, vol. 34, 2022, Art. no. e6683.

[31] L. Nardi, A. Souza, D. Koeplinger, and K. Olukotun, "HyperMapper: A practical design space exploration framework," in *Proc. IEEE 27th Int. Symp. Model. Anal. Simul. Comput. Telecommunication Syst.*, 2019, pp. 425–426.

[32] E. O. Hellsten et al., "BaCO: A fast and portable Bayesian compiler optimization framework," in *Proc. 28th ACM Int. Conf. Architectural Support Program. Languages Operating Syst.*, 2024, pp. 19–42.

[33] F.-J. Willemsen, R. van Nieuwpoort, and B. van Werkhoven, "Bayesian optimization for auto-tuning GPU kernels," in *Proc. Int. Workshop Perform. Model. Benchmarking Simul. High Perform. Comput. Syst.*, 2021, pp. 106–117.

[34] C. Rubio-González et al., "Precimonious: Tuning assistant for floating-point precision," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2013, pp. 1–12.

[35] W.-F. Chiang et al., "Rigorous floating-point mixed-precision tuning," in *Proc. 44th ACM SIGPLAN Symp. Princ. Program. Languages*, 2017, pp. 300–315.

[36] A. Solovyev et al., "Rigorous estimation of floating-point round-off errors with symbolic Taylor expansions," *Trans. Program. Languages Syst.*, vol. 41, pp. 1–39, 2018.

[37] H. Menon et al., "ADAPT: Algorithmic differentiation applied to floating-point precision tuning," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2018, pp. 614–626.

[38] M. O. Lam et al., "Automatically adapting programs for mixed-precision floating-point computation," in *Proc. 27th Int. ACM Int. Conf. Supercomputing*, 2013, pp. 369–378.

[39] Y. Wang and C. Rubio-González, "Predicting performance and accuracy of mixed-precision programs for precision tuning," in *Proc. Int. Conf. Softw. Eng.*, 2024, pp. 1–13.

[40] S. Heldens and B. van Werkhoven, "Kernel float: Unlocking mixed-precision GPU programming," *ACM Trans. Math. Softw.*, New York, NY, USA: Dec. 2025. [Online]. Available: https://doi.org/10.1145/3779120.

[41] M. Lurati et al., "Bringing auto-tuning to HIP: Analysis of tuning impact and difficulty on AMD and Nvidia GPUs," in *Proc. Eur. Conf. Parallel Process.*, 2024, pp. 91–106.

[42] C. A. Coello Coello, "Theoretical and numerical constraint-handling techniques used with evolutionary algorithms: A survey of the state of the art," *Comput. Methods Appl. Mechan. Eng.*, vol. 191, pp. 1245–1287, 2002.

[43] R. A. Schoonhoven, B. van Werkhoven, and K. J. Batenburg, "Benchmarking optimization algorithms for auto-tuning GPU kernels," *IEEE Trans. Evol. Comput.*, vol. 27, no. 3, pp. 550–564, Jun. 2023.

[44] J. O. Tørring, B. van Werkhoven, F. Petrovč, F. -J. Willemsen, J. Filipovič, and A. C. Elster, "Towards a benchmarking suite for kernel tuners," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops*, 2023, pp. 724–733.

[45] S. Che et al., "Rodinia: A benchmark suite for heterogeneous computing," in *Proc. Int. Symp. Workload Characterization*, 2009, pp. 44–54.

[46] A. Yazdanbakhsh, D. Mahajan, H. Esmaeilzadeh, and P. Lotfi-Kamran, "AxBench: A multiplatform benchmark suite for approximate computing," *IEEE Des. Test*, vol. 34, no. 2, pp. 60–68, Apr. 2017.

[47] J. A. Stratton et al., "Parboil: A revised benchmark suite for scientific and commercial throughput computing," University of Illinois, Urbana–Champaign Mar. 2012. http://impact.crhc.illinois.edu/Shared/Docs/impact-12-01.parboil.pdf

[48] NVIDIA, "CUDA Samples," Version 13.1, Samples for CUDA Toolkit 2025. https://github.com/NVIDIA/cuda-samples

[49] H. Bal et al., "A medium-scale distributed system for computer science research: Infrastructure for the long term," *Computer*, vol. 49, no. 5, pp. 54–63, May 2016.

[50] LUMI Consortium, "LUMI supercomputer," 2024. [Online]. Available: https://www.lumi-supercomputer.eu/

[51] F.-J. Willemsen et al., "Efficient construction of large search spaces for auto-tuning," in *Proc. Int. Conf. Parallel Process.*, 2025, pp. 668–677.

[52] F.-J. Willemsen, R. V. van Nieuwpoort, and B. van Werkhoven, "Tuning the tuner: Introducing hyperparameter optimization for auto-tuning," in *Proc. IEEE Int. Conf. eScience*, 2025, pp. 213–222.

[53] F.-J. Willemsen, N. van Stein, and B. van Werkhoven, "Automated algorithm design for auto-tuning optimizers," 2025, *arXiv:2510.17899*.

[54] M. Binois and N. Wycoff, "A survey on high-dimensional Gaussian process modeling with application to Bayesian optimization," *ACM Trans. Evol. Learn. Optim.*, vol. 2, pp. 1–26, 2022.

[55] M. B. Salem et al., "Gaussian process-based dimension reduction for goal-oriented sequential design," *SIAM/ASA J. Uncertainty Quantification*, vol. 7, pp. 1369–1397, 2019.

[56] Y. Wei et al., "Scalable Bayesian optimization via focalized sparse gaussian processes," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2024, pp. 120443–120467.

[57] Y. Diouane et al., "TREGO: A trust-region framework for efficient global optimization," *J. Glob. Optim.*, vol. 86, pp. 1–23, 2023.

[58] Y. Jin, "Surrogate-assisted evolutionary computation: Recent advances and future challenges," *Swarm Evol. Computation*, vol. 1, pp. 61–70, 2011.

[59] S. V. D. Vlugt et al., "PowerSensor3: A fast and accurate open source power measurement tool," in *Proc. Int. Symp. Perform. Anal. Syst. Softw.*, 2025, pp. 213–226.

[60] A. Banerjee et al., "A generalized maximum entropy approach to Bregman co-clustering and matrix approximation," in *Proc. ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2004, pp. 509–514.

[61] X. Wu et al., "A novel analysis of spring phenological patterns over Europe based on co-clustering," *J. Geophysical Res. Biogeosciences*, vol. 121, pp. 1434–1448, 2016.