MLKAPS: Machine Learning and Adaptive Sampling for HPC Kernel Auto-tuning

Mathys Jam, Eric Petit, Pablo de Oliveira Castro, David Defour, Greg Henry, William Jalby

MLKAPS: Machine Learning and Adaptive Sampling for HPC Kernel Auto-tuning

MATHYS JAM, Université Paris-Saclay, UVSQ, LI-PaRAD, France
ERIC PETIT, Intel Corp., USA
PABLO DE OLIVEIRA CASTRO, Université Paris-Saclay, UVSQ, LI-PaRAD, France
DAVID DEFOUR, Université de Perpignan via Domitia, UPVD, LAMPS, France
GREG HENRY, Intel Corp., USA
WILLIAM JALBY, Université Paris-Saclay, UVSQ, LI-PaRAD, France

Many High-Performance Computing (HPC) libraries rely on decision trees to select the best kernel hyperparameters at runtime, depending on the input and environment. However, finding optimized configurations for each input and environment is challenging and requires significant manual effort and computational resources. This paper presents MLKAPS, a tool that automates this task using machine learning and adaptive sampling techniques. MLKAPS generates decision trees that tune HPC kernels' design parameters to achieve efficient performance for any user input. MLKAPS scales to large input and design spaces, outperforming similar state-of-the-art auto-tuning tools in tuning time and mean speedup. We demonstrate the benefits of MLKAPS on the highly optimized Intel MKL `dgetrf` LU kernel and show that MLKAPS finds blindspots in the manual tuning of HPC experts. It improves over 85% of the inputs with a geomean speedup of ×1.30. On the Intel MKL `dgeqrf` QR kernel, MLKAPS improves performance on 85% of the inputs with a geomean speedup of ×1.18.

CCS Concepts: • **Software and its engineering** → **Software evolution**; **Software performance**; • **Computing methodologies** → *Boosting*.

Additional Key Words and Phrases: Autotuning, input-aware optimization, Math Library

ACM Reference Format:
Mathys Jam, Eric Petit, Pablo de Oliveira Castro, David Defour, Greg Henry, and William Jalby. 2024. MLKAPS: Machine Learning and Adaptive Sampling for HPC Kernel Auto-tuning. 1, 1 (January 2024), 19 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

High-Performance Computing (HPC) engineers have adopted strategies to ensure that kernels perform well in different environments by configuring and dynamically selecting the best implementation at runtime, depending on the user's input and execution context. Software that uses decision trees [18], such as the Intel Math Kernel Library (MKL)[16], uses an internal set of predefined optimized configurations of internal parameters. Indeed, kernels rely on two sets of runtime parameters: input parameters describing the data and arguments from the user, and design parameters selecting different algorithmic versions or runtime behavior. We aim to efficiently explore and generate the selection

Authors' addresses: Mathys Jam, mathys.jam@uvsq.fr, Université Paris-Saclay, UVSQ, LI-PaRAD, France, Guyancourt; Eric Petit, eric.petit@intel.com, Intel Corp., Oregon, USA, Hillsboro; Pablo de Oliveira Castro, pablo.oliveira@uvsq.fr, Université Paris-Saclay, UVSQ, LI-PaRAD, France, Guyancourt; David Defour, david.defour@univ-prerp.fr, Université de Perpignan via Domitia, UPVD, LAMPS, France, Perpignan; Greg Henry, greg.henry@intel.com, Intel Corp., Oregon, USA, Hillsboro; William Jalby, william.jalby@uvsq.fr, Université Paris-Saclay, UVSQ, LI-PaRAD, France, Guyancourt.

Manuscript submitted to ACM

mechanism that associates a design configuration with any given input parameters. Increasing the number of available configurations and the complexity of the selection mechanism improves the performance and adaptability of the kernel at the expense of runtime decision cost and represents a complex trade-off for optimization.

Furthermore, the curse of dimensionality [30] makes it exponentially harder to find good design configurations as the size of the input and the design space increases. Recent hardware architectures with many cores, different vector sizes, numa effects, and the use of GPU accelerators contribute to larger design spaces. Additionally, solutions relying on manual exploration are subject to human bias and lead to blind spots (regions of the input space where the configuration is sub-optimal due to an unexplored combination of parameters). As an exhaustive search becomes intractable for large design spaces, alternative solutions must be found for efficient exploration. In the `dgetrf` kernel (LU) from Intel MKL, the design space contains 4.6×10^{13} possible internal configurations and a target region of interest of 1.6×10^{7} possible inputs (more details in Sec. 5.3). This is a key limitation for the scalability of sampling based auto-tuning methods that we address in this work. To address this, we explore a new adaptive sampling heuristic.

This article describes MLKAPS (Machine Learning for Kernel Accuracy and Performance Studies), an open-source auto-tuning framework targeting high-performance software design problems. MLKAPS automatically generates decision trees to map every input to optimized design parameters at runtime. The proposed methodology is hardware-agnostic and does not make assumptions about the optimization objectives. The former can be the kernel's execution time, numerical accuracy, or energy, depending on the user's needs. In this paper, though, we are focusing on execution time. The tool is released under an open-source license at https://github.com/MLCGO/MLKAPS. This tool can benefit many high-performance libraries, as we demonstrate on three kernels from the Intel Math Kernel Library and ScaLAPACK.

Our contributions and key results are the following:

- The MLKAPS framework, an integrated workflow to automatically generate decision trees to select optimized design parameters for any given set of inputs. Its only inputs are a description of the parameters and a kernel to evaluate configurations.
- We describe an implementation of statistical adaptive sampling (based on space-filling designs and variance-based techniques [11]) and a new sampling strategy we developed, GA-Adaptive, to overcome the high-dimensional space exploration problem. We demonstrate that GA-Adaptive outperforms other competitive sampling strategies for kernel optimization on large input regions.
- We demonstrate the benefits of the decision trees generated by MLKAPS on two different hardware architectures, using high-performance kernels from Intel MKL: `dgetrf` (LU) and `dgeqrf` (QR). Those kernels were highly tuned by the MKL team, and offer complex objective spaces representative of real industrial applications.
- While sampling a very small fraction ($\approx \frac{1}{10^{10}}$) of the total design space, MLKAPS improves Intel MKL `dgetrf` performance on 85% of the inputs by an geometric mean speedup of 30%. The mean regression (slowdown) is 0.90 and 1.46 for progressions (speedup > 1.0). Furthermore, the quality of MLKAPS results continuously improves with the number of samples, and the resulting design configurations and speedup are not the same for the two architectures used, showcasing that MLKAPS adapts to its environment.
- We compare MLKAPS with the well-known Optuna [1] autotuner on the Intel MKL `dgeqrf` (QR) kernel. MLKAPS obtains a geomean speedup of 36% over Optuna.
- We compare MLKAPS to GPTune [20], a state-of-the-art autotuner, and show that MLKAPS has better performance on their ScaLAPACK use-case and our case of Intel MKL `dgetrf` (LU). The MLKAPS pipeline also scales to a large sample count with relatively low overhead compared to GPTune.

```
double sum(double* matrix,
           int n, int m, int T) {
  double sum = 0;
  #pragma omp parallel for \
  reduce(+: sum) num_threads(T)
  for (int i = 0; i < n*m; i++)
    sum += matrix[I];
  return sum;
}
```

Fig. 1. **Illustrative kernel summing the elements of a matrix** with three input parameters (`matrix`, `n`, `m`), and one design parameter (`T`).

```
double sum_tuned(double* matrix,
           int n, int m) {
    // T (number of threads)
    // is tuned depending
    // on the input parameters
    int T = decision_tree(n, m);
    return sum(matrix, n, m, T);
}
```

Fig. 2. **MLKAPS generates a tuning decision tree** to select the best number-of-threads `T` for a given input context.

This paper is organized as follows: In Section 2, we define the optimization problem in large dimensional space that MLKAPS aims at solving. In Section 3, we review state-of-the-art auto-tuning approaches and discuss how they compare to MLKAPS. In Section 4, we detail MLKAPS design principles, pipeline, and the adaptive sampling methods we explored to improve the scalability and final optimization results. In Section 5, we demonstrate the scalability and tuning benefits of using MLKAPS on highly hand-optimized versions of HPC kernels. Sections 6 and 7 conclude and give insight into future works.

2 PROBLEM DEFINITION

To illustrate the kind of tuning problems we address, let us consider a naive `OpenMP` [25] kernel, which sums every element of an input matrix (Figure 1). The execution of this kernel is affected by various parameters, which can be classified as input and design parameters. In practice, such parameters can be of different types, such as real, integer, categorical, or boolean. As illustrated in Figure 2, the MLKAPS objective is to produce a decision tree that selects the best design parameters (`T`) according to the input parameters (`n`, `m`). The decision trees are generated as C code to be embedded and shipped with the kernel for predictions at runtime.

Input parameters correspond to the task specification and cannot be tuned. They are either defined by the user (e.g., `matrix` values, `n`, and `m`) or correspond to fixed parameters of the execution environment (e.g., available memory, available number of cores.)

Design parameters refer to the set of parameters MLKAPS optimizes. Unique combinations of design parameter values are referred to as *configurations*. These parameters may be internal to the kernel (e.g. algorithmic variant, number `T` of threads in the previous example) or the execution environment (e.g., thread placement, core frequency, compiler flags.)

The decision tree provides optimized values while minimizing execution time or any other objective defined by the user. This decision tree is evaluated at runtime to select the best configuration depending on the input(s).

3 RELATED WORKS

In [24], the authors survey the field of auto-tuning programs and classify the different frameworks based on their approach used for each different component of the tuning problem they are addressing:

(A) Tuning rating method (how are solutions evaluated): empirical evaluation or model-driven.
(B) Tuning time (when does the tuning occur): online tuning, offline tuning
(C) Search space exploration algorithm: (biased) random search, brute-force, parallel search, profile-based optimization, and pruned search space tuning
(D) Tuning scope: program-level, section-level, procedure-level

Since MLKAPS operates in a black-box setup, it is kernel-agnostic and can target all three tuning scope levels mentioned in (D). In the following, we review the related works using the criteria (A), (B), and (C).

3.1 Tuning rating method

Tuning rating is the method used to evaluate a configuration and how it compares to others. On one hand, Atlas [31], Spiral [26], and FFTW [13] are auto-tuned libraries that use empirical rating: the configurations are executed in the target environment, which is costly but also brings the most information. More recently, Bolt [32] and AutoTVM [9] also use empirical evaluation to evaluate a set of candidate configurations for Deep Neural Network optimization.

On the other hand, Bergstra et al. [6] implement a model-driven strategy: an approximation of the kernel's objectives is used to rank the tuning solutions. This surrogate model can be built in two ways: using heuristics/models based on expert knowledge of the kernel or ML models trained on the kernel. Using a model allows the optimization process to rate as many configurations as needed. Hand-tuned cost models are error-prone and subject to bias but are relatively cheap to produce. AutoTVM and Ansor [19] build Gradient Boosting Decision Trees (GBDT) [2], a machine learning ensemble method based on decision trees, to perform a preliminary low-cost evaluation of the solutions.

3.2 Tuning time

In Atlas [31] or Patus [10], the tuning process occurs offline before the user's program is run. By reloading the optimized solutions, runtime overhead is minimal. However, offline tuning can suffer from a lack of contextual information available at runtime.

In online tuning, the optimization process occurs at runtime with contextual knowledge. In [28], hill-climbing is used at runtime in a model-driven approach. In [14], multiple tools monitor and adapt the kernel tuning to honor specific application requirements. The difficulty in this approach is the trade-off between optimization overhead and the quality of the tuning results: the optimization cost must not overshadow the benefits realized in application performance.

In Bolt [32], domain-specific knowledge, hardware information, and heuristics are used to reduce the tuning time to around 20 minutes per tuning iteration. This is a compromise between black box tuners, which can run for days, and hand-tuned cost model approaches. This is particularly useful for their Deep Neural Network target applications, as users can try different neural network layouts in quick iterations. Felix [33] proposes a similar approach, using pre-trained models as predictor using the Tenset [34]. They evaluate models such as MLP and boosting trees to predict kernels' performance based on their features (number of load, store, reuse distance…) and hardware features (cache sizes…) to select candidate loop scheduling in tensor computation. While using a predictor handling large range of input characteristics, the prediction and final exploration are too expensive for runtime evaluation, and requires kernel bounds to be known at compile time. This perfectly fit the JIT use case they aim at, but cannot cover the requirement of HPC applications such as math library. However, this could be used to tune lower level kernels used in the math libraries algorithm.

3.3 Exploration algorithm

The last criterion is the algorithm used to explore the search space. Atlas and many other auto-tuned libraries rely on brute-force search with an empirical rating. Adaptive sampling strategies fall in the biased random search category, which covers a wide range of techniques. They typically start with a random walk in the search space before introducing a bias towards the most promising solutions. Optuna [1] uses a combination of Covariance Matrix Adaptation Evolution Strategy (CMA-ES) [15] and Tree-structured Parzen Estimator (TPE) [5] to explore the design space, using empirical evaluations paired with an early-stopping criterion that stops trials that perform worse than the best-known solution. We compare MLKAPS and Optuna in Section 5.4.1.

Luszczek et al. [21] uses a Bayesian optimization technique combined with multi-task and transfer learning to solve an auto-tuning problem. GPTune [20] builds Gaussian-Process (GP) models in a Bayesian optimization pipeline. However, while most work presented here only outputs a unique configuration for the application, GPTune works on a problem similar to MLKAPS, mapping multiple inputs to dedicated configurations. Furthermore, while MLKAPS uses decision trees to generalize, GPTune implements a strategy called TLAII, which can infer configurations for new inputs without resampling, based on previous results. We compare MLKAPS and GPTune in Section 5.4.3.

In Bolt [32], and AutoTVM [9] authors use domain-specific knowledge to inform their search. Bolt uses hardware information to restrain the space of potential configurations based on a set of rules for Nvidia GPUs. AutoTVM uses an iterative algorithm based on GBDT [2] and simulated annealing [27].

4 MLKAPS DESCRIPTION

MLKAPS (Machine Learning for Kernel Accuracy and Performance Studies) uses a model-driven rating method powered by GBDT models from the LightGBM [17] Python library trained using empirical evaluations. It was designed to perform equally well on all regions of the input space, leverage transfer learning to increase efficiency by sharing knowledge across regions, and scale to large design spaces. Figure 3 shows the pipeline implemented in MLKAPS.

We divide MLKAPS' pipeline into two components: first, sampling and modeling, detailed in Section 4.1, where we collect knowledge about the kernel and refine the GBDT model; and second, optimization and decision tree creation, detailed in Section 4.2, where we use the surrogate model to find a set of optimized configurations.

4.1 Sampling and Modeling

Let us assume that we are initially given a black-box kernel that measures the target objective for any given inputs and design parameters. We aim to gather a dataset to train a machine-learning model to infer the objective value. We will refer to a kernel execution as a sample. Finally, we will use this as a surrogate model for configuration rating, allowing us to replace costly kernel sampling with the low-cost inference of the model.

As measuring every possible configuration is intractable due to the combinatorial explosion and sample execution cost, the measured samples must be carefully selected. We aim to maximize how much useful information the surrogate model can extract from each sample towards our optimization objective. We implemented various adaptive sampling approaches, trading between exploring new input areas and exploiting current knowledge to determine the next samples of interest. MLKAPS implements three main strategies described in the next sections: space-filling sampling, variance-based sampling, and optimization-driven sampling. Those strategies are illustrated in Figure 3, and will be compared in Section 5.

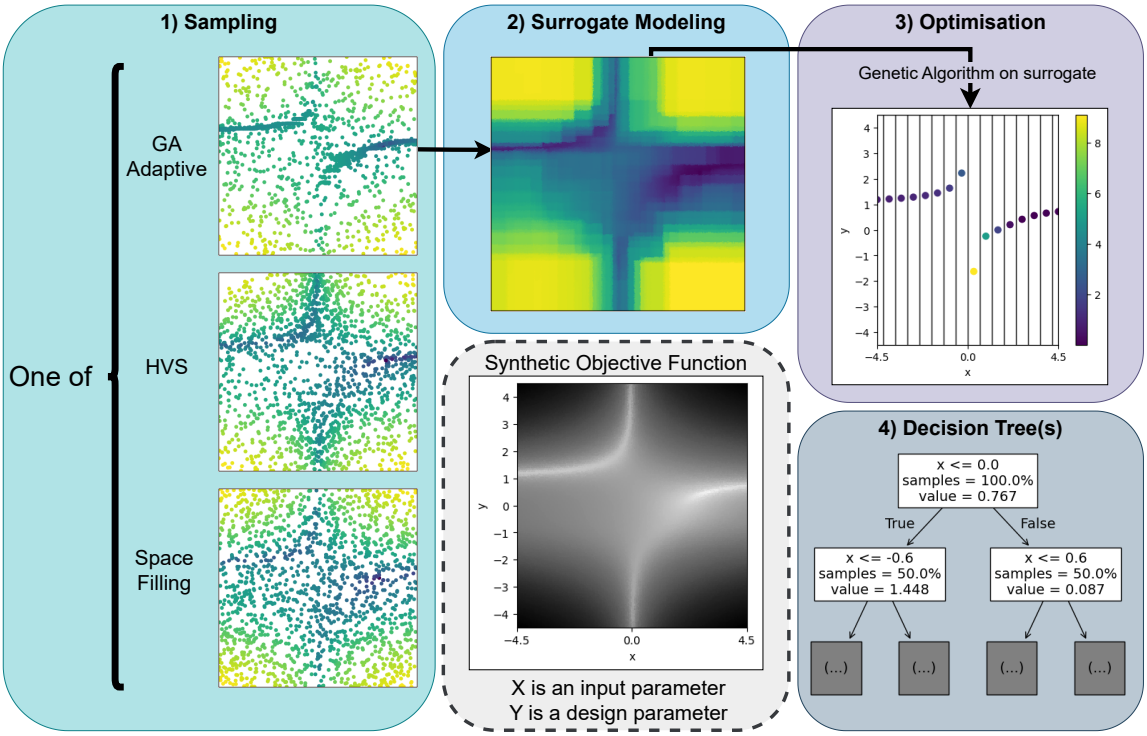

Fig. 3. **Flowchart of the MLKAPS auto-tuning pipeline.** This example illustrates the different steps to build a decision tree that maps one design parameter Y to one input parameter X. For an industrial application with a much larger number of parameters, it would be intractable to map the objective function exhaustively as shown here.

4.1.1 Space-filling sampling. is the simplest method for sampling points spread over the entire input space. It aims to bias for spatial coverage, as shown in figure 3, by ensuring that all configurations have an equal probability of being sampled. A simple example is random sampling. Latin Hypercube Sampling (LHS) [22] is an alternative sampling method that divides each parameter into intervals and ensures that each interval is sampled exactly once. This approach provides better coverage of each of the parameter space and captures variations and interactions more effectively, which can help reduce uncertainty arising from measurement noises.

4.1.2 HVS sampling. is another strategy to characterize a function efficiently by biasing the sampling based on variance. In [11], authors propose an iterative sampling strategy based on a partitioning of the parameter space, named Hierarchical Variance Sampling (HVS). LHS is used for bootstrapping the method. The samples are then partitioned using a decision tree, and the partition size and variance estimator are computed. The next iteration of sample selection makes a trade-off between exploiting current variance estimator information and exploring new regions, therefore, the samples are distributed among the partitions depending on the product size $\times$ variance. In figure 3, some regions have a very low sample density due to their low variance, which is better redistributed to characterize complex regions of the objective function.

The variant HVS-relative (HVSr) [11] uses an estimator of the coefficient of variation for cases where the objective function has a large range of non-uniformly distributed values.

input : b ratio of initial bootstrap samples
i the initial ratio of points taken with the GA sampler
f the final ratio of points taken with the GA sampler
s the number of samples to take per iteration
n the total number of samples

output : set of sampled points

1 $Samples \leftarrow$ **BootstrapLHS**$(b \times n)$; // Bootstrap the algorithm with $b \times n$ LHS samples
2 **while** $|Samples| < n$ **do**
3 $p \leftarrow |Samples|/n$;
4 $\epsilon \leftarrow i + (f - i) \times p$;
5 $Model \leftarrow$ **GBDT**$(Samples[Parameters], Samples[Objective])$; // Fit a surrogate model
6 $OptimPoints \leftarrow$ **PickRandomInputs**$(\epsilon \times s)$; // Select random inputs for GA
7 $New_{ga} \leftarrow$ **GA**$(OptimPoints, Model)$; // Exploitation: Optimize with GA on $\epsilon \times s$ new inputs
8 $New_{sub} \leftarrow$ **SubSampler**$((1 - \epsilon) \times s, Samples)$; // Exploration: Pick leftover samples with sub-sampler
9 $Samples \leftarrow Samples \cup New_{sub} \cup New_{ga}$;
10 **end**
11 return $Samples$;

Fig. 4. **GA-Adaptive core loop**. This algorithm uses a strategy similar to epsilon-decreasing [3] to solve the exploration-exploitation dilemma by linearly increasing the number of samples taken for exploitation. Note how the sampling-modeling-optimization steps are similar to the global pipeline. At each iteration, the percentage of points taken with GA and the sub-sampler is computed as a linear interpolation between the initial and final ratios, controlled by the completion percentage (i.e., at 50% completion with an initial ratio of 0 and a final ratio of 0.8, GA will pick 40% of points in the next iteration).

Furthermore, some ill-configurations can give really poor execution time and introduce high-variance regions in the objective space. As a consequence, HVS can spend a huge portion of its sampling budget trying to capture such configurations, while not providing useful information on the potential local optima. The method does not distinguish between variance introduced by good and bad configurations. We introduced an upper bound in the objective function to prevent HVS from overly-sampling outlier configurations. This proved very effective in the case of `dgetrf` (LU) and `dgeqrf` (QR), with negligible quantities of wasted samples in our final set of experiments.

4.1.3 GA-adaptive sampling. We develop a new optimization-driven sampling strategy, GA-adaptive, with the rationale that the surrogate model does not need to learn the entire objective space; it should trade off generalization for high accuracy in regions containing good configurations. This is visible in figure 3, where almost no points are chosen outside the optimal neighborhoods, trading global accuracy for better predictions on optimized configurations.

It deals with the exploration/exploitation dilemma [29] by replicating the MLKAPS' optimization phase using an ϵ-decreasing strategy [3].

Algorithm 4 shows a pseudo-code of GA-Adaptive. The algorithm is initialized (Line 1) using LHS to take the initial points and gain enough knowledge to build a surrogate for the GA sampler.

During each sampling iteration, a surrogate model is fitted on past samples. Input configurations are randomly selected and optimized using a Genetic Algorithm on the surrogate model. The more the algorithm samples, the more accurate the model will be on good configurations. Therefore, the GA runs will likely converge towards the optima thanks to the following effects:

- If the model is overly optimistic on a configuration, it will be selected for sampling and the true value added to the dataset, correcting the model so that future iterations will avoid it. This also applies to the final optimization phase in MLKAPS' pipeline, as it uses the same modeling and optimization algorithms.
- If the model prediction was right, the local accuracy of the surrogates around promising solutions is improved. This allows us to differentiate between two solutions with similar performance, reducing uncertainty caused by noise.

In the first iterations, exploration is favored at the expense of exploitation to consolidate the surrogate model. Then, as the model accuracy improves over time, we move towards exploitation and focus on improving knowledge of good configurations. This proportion follows a linear progression between user-defined starting and end bounds according to the iteration count.

GA-Adaptive behavior depends on a set of hyper-parameters: the algorithmic choice for the optimization process (GAs in our implementation), the bootstrapping algorithm (LHS), the sub-sampler (HVSr by default), and the ratio of points taken with GA (ϵ), usually starting at 0.0 with a final value of 1.0 at the last iteration.

4.1.4 Modeling. is the last step of the first stage, where we train the main surrogate model using the previously sampled dataset. we focused on gradient-boosting decision trees (GBDT) [2] from `LightGBM` [17]. It allows efficient handling of all variable types, including categorical, and offers scalable and accurate modeling. GBDT is used by many existing autotuners [6, 9]

Plots 6 and 7 show different model metrics have different convergence behavior, making it challenging to select the most suitable metric for our optimization objective. Since we have no prior knowledge of the objective space, we usually select Mean Absolute Error (MAE) for the model's metric, which works well in the cases studied in this paper. In cases where the objective value can cover a huge range of values, Mean Average Percentage Error (MAPE) was observed to improve the tuning results significantly. However, the only reliable indicator of model performance we have available is comparing speedup or mean performance using the solutions the optimizer found on the surrogate. Those metrics were the most helpful in our development of the MLKAPS pipeline.

We used two approaches to tune the surrogate's hyperparameters: manual tuning and Optuna [1]. Using Optuna with an MAE metric led to worse optimization than a hand-tuned model. We believe this is due to the need for a dedicated metric for the model, one that measures the accuracy on the best configurations. Given the impact of surrogate quality on our result, we are considering defining and experimenting with such new metrics in future work.

4.2 Optimization and Decision trees

This step computes a set of optimized configurations across the input space using GAs. We use `pymoo`'s [22][8] implementation of NSGAII [12] for its robustness and generalization capabilities. We first select "optimization points" by building a regular grid over the input space, with a user-defined size in each dimension. We then run one GA instance per point in the grid and find the best configuration, one that leverages the local behavior of the objective function on a given input region. Optimal performance in HPC usually occurs on cliffs [4], dictated by cache size, pipeline saturation, or other discrete thresholds. As such, the optimization quality is significantly impacted by the coarseness of the grid. A coarse grid might lead to suboptimal choices, while a finer grid better leverages locality. There is a trade-off between the size of the grid and the time budget for each point. Running the optimization phase on a surrogate model with cheap predictions allows us to increase the density of the grid. And since MLKAPS sampling is not restricted by the grid, the grid point configurations will be informed by their local neighborhood and not limited by strictly local information,

Fig. 5. **Hardware architectures used for the experiments.**

	CPU	freq.	cores	threads	L1	L2	L3	Ram	Type
KNM	Intel Knights Mill	1.5GHz	72	288	32 KB	36 MB		16GB	HBM
SPR	Intel Xeon Gold 6438M	2.2GHz	64	128	80 KB	2 MB	60MB	504GB	DDR5

resulting in a better generalization of the solution and limiting potential over-fitting. In practice, we did not observe improvement by selecting 24x24 grid compared to 16x16 and as a consequence select the later as default value for our experiment unless specified otherwise.

The final step of the pipeline is to build one or more decision trees on the previous grid. For this, we use `scikit-learn`'s Decision Tree Regressor for continuous variables and classifier for categorical variables. Those trees are stored in a pickled file, and generated as C code for the user to embed in his kernel. We currently build one decision tree per design parameter, whose outputs are combined to get the full configuration.

As the runtime overhead of the decision process could take away the potential gains, the application often has a limited time budget to decide the configuration to use. Deep trees maximize choice locality but can have a significant overhead; shallow trees will select a trade-off between multiple configurations, reducing the overhead at the cost of suboptimal configurations.

5 RESULTS

5.0.1 Experimental Setup. We evaluated MLKAPS on two HPC servers with widely different micro-architectures: an Intel Knight Mill processor and Intel Sapphire Rapids processor, described in Table 5 and studied three high-performance kernels:

- `dgetrf` and `dgeqrf` from Intel MKL. The `dgetrf` and `dgeqrf` binary Intel provided us is statically linked to a prototype derived from MKL version 2022.0 Update 2, OpenMP 4.5, and contained the kernel version specifically tuned for both the KNM and SPR. Furthermore, we explore the input and design parameters range specified by Intel for this benchmark.
- `pdgeqrf` from ScaLAPACK [7]. We use ScaLAPACK Version 2.1.0 combined with OpenMPI 4.0 as provided in the GPTune Docker image[1].

5.0.2 `dgetrf` *description.* We focus on a detailed experiment and analysis on Intel MKL `dgetrf` kernel. This kernel performs an LU factorization based on level 3 BLAS computation. Two of its input parameters correspond to the matrix size, named n and m, which significantly influence the execution time, as the flop count is approximately $\frac{2}{3}m^2n$ (depending on the largest dimension). We bound $1000 \leq n, m \leq 5000$ for this study according to the benchmark specification. The kernel exposes eight internal design parameters, such as the number of threads and tilling configuration, which lead to up to ten dimensions to explore during the sampling phase. When we compare to the MKL, we consider their current hand-tuned internal configurations as a reference. Recall that MLKAPS is a black-box approach: it assumes no prior knowledge about these parameters and their role, and as such, we did not reuse any information that could be extracted from this hand-tuning. This study considers a single objective: minimizing execution time. MLKAPS uses depth 8 decision trees trained on an optimization grid 16×16.

[1] liuyangzhuan/gptune:4.5 https://hub.docker.com/r/liuyangzhuan/gptune

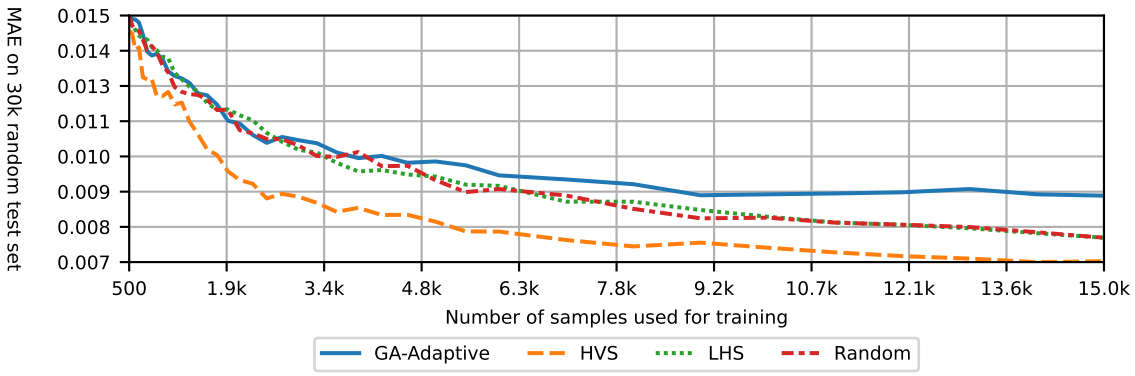

Fig. 6. **(SPR): Global accuracy measured using 30k random samples with GBDT surrogate models trained using different sampling strategies.** (lower is better). Up to 15k samples per method, same model hyperparameters. HVS outperforms the other samplers for global accuracy; thus, it is a good fit for the exploration phase.

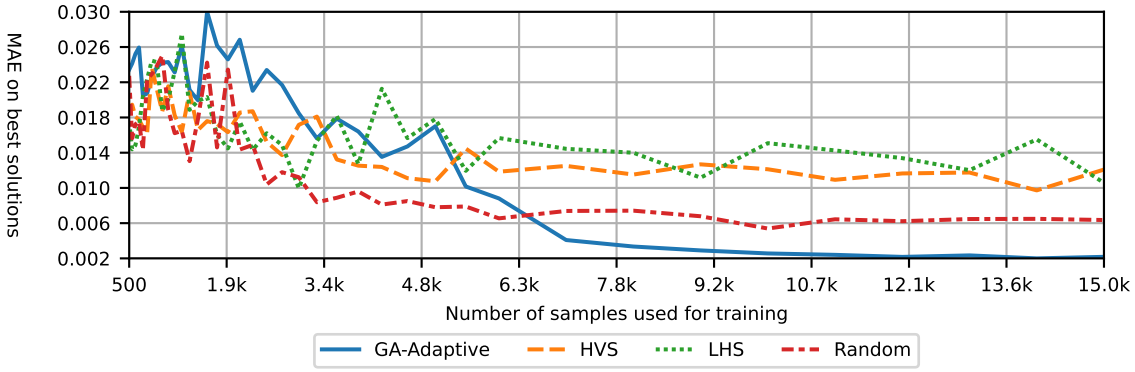

Fig. 7. **(SPR): Local accuracy on the predicted best configurations (1024 per method) with GBDT surrogate models trained using different sampling strategies.** (lower is better). Up to 15k samples per method, same model hyperparameters. GA-Adaptive outperforms all methods when optimizing for local accuracy.

First, Section 5.1 compares the different sampling strategies and their impact on the accuracy of the GBDT model. Section 5.2 evaluates the tuning capability of each algorithm. Section 5.3 demonstrates that MLKAPS improves the existing tuning of Intel `dgetrf` on both architectures. Section 5.4.3 compares MLKAPS with the state-of-the-art auto-tuners, GPTune [20] and Optuna [1].

5.1 Sampling strategies and model accuracy

Since optimization runs on top of the surrogate model, its accuracy is critical and must be carefully evaluated for each sampling algorithm. We ran the different algorithms on Intel MKL `dgetrf` kernel with $15k$ samples and evaluated the accuracy of the model. This experiment focuses on comparing and validating our heuristic for the GA-adaptive algorithm.

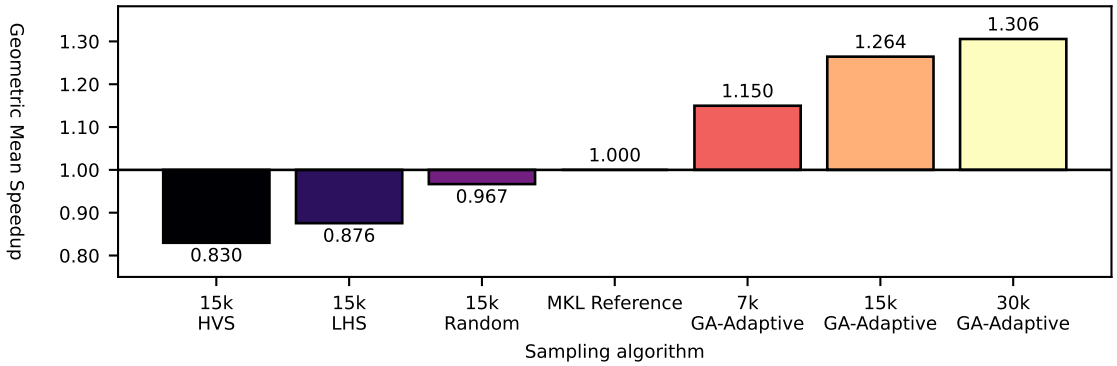

Fig. 8. **(SPR): Geometric mean Speedup (higher is better) against the MKL reference configuration on `dgetrf` (LU)**, depending on the sampling algorithm. 46x46 validation grid. 7k/15k/30k denotes the samples count. GA-Adaptive outperforms all other sampling strategies for auto-tuning. With 30k samples it achieves a mean speedup of ×1.3 of the MKL dgetrf kernel.

5.1.1 Global accuracy. First, we measured the global precision of the model using a validation data set composed of 30k random samples. Figure 6 shows the global precision of the resulting models. We observe that HVS outperforms the other strategies, which is expected as it was developed to maximize the accuracy gained with each sample. LHS and Random have similar performance, while GA-Adaptive has a higher error, which is expected since it sacrifices global accuracy.

5.1.2 Local accuracy. Then, we measured the local accuracy of each method on the output of the optimization phase with the same model. This gives us the accuracy around the predicted optimal configurations. Figure 7 shows that GA-Adaptive has a significantly lower MAE on the best solutions, meaning it chooses the best configurations with higher confidence.

5.2 Sampling strategies and optimization

To compare the sampling strategies implemented in MLKAPS, we built a regular grid of 46×46 points (for a total of 2116 points) in the input space. We then measured the configuration predicted by MLKAPS with different sampling strategies and sample counts and computed the geometric mean speedup relative to the MKL configuration.

The result given in Fig. 8 and previous observations on the influence on local and global accuracy of sampling strategies endorse our heuristic that improving local accuracy using GA-Adaptive is more beneficial to tuning quality than improving the overall accuracy using HVS. It also shows that measuring the global accuracy using MAE or RMSE on random samples does not allow us to conclude on the model's quality for tuning, as GA-Adaptive performed worse than random sampling on this metric. While HVS produced the best model for global accuracy, it performs worse than random sampling for tuning this kernel.

5.3 Improvment upon hand-tuning

In the following, we will dive deeper into how tuning solutions obtained with MLKAPS perform when compared to the highly optimized hand-tuning of the Intel MKL `dgetrf` kernel.

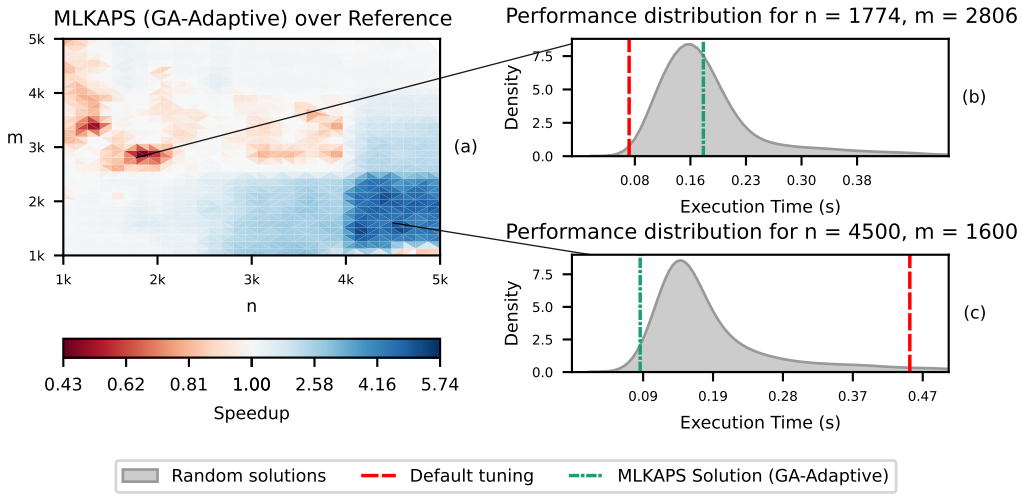

Fig. 9. **(KNM): (a) Speedup map of GA-Adaptive** (7k samples) over the Intel MKL hand-tuning for `dgetrf` (LU), higher is better. **(b) Analysis of the slowdown region** (performance regression). **(c) Analysis of the high speedup region.** 3,000 random solutions were evaluated for each distribution.

5.3.1 Results on the KNM. Figure 9 (a) shows the relative performance of GA-Adaptive compared to the MKL tuning on the KNM architecture, using a 32×32 regular grid over the input space. For this experiment, we limit the number of samples to 7000 to keep the runtime short enough to explore MLKAPS configurations and illustrate pitfalls of not having enough samples. We observe that MLKAPS's results are non-uniform over the input space:

- The first region, with $2500 < m < 5000$, shows an average speedup of 1, meaning that MLKAPS performs as well as the MKL reference. However, for some inputs, MLKAPS shows significant performance regressions.
- The second region, with $1000 \leq m \leq 2500$ shows significant speedups for MLKAPS, up to $\times 5$ for $n > 4000$
- The grid-like pattern is due to the decision trees partitioning of the input space.

To understand the speedup map, we chose one point in the best and worst regions and measured randomly chosen configurations. This stochastic sampling allows us to estimate the performance histogram in the region and see where MLKAPS and MKL choices lie in the distribution. The first Histogram 9 (b) shows the distribution of performance for $(n = 1774; m = 2806)$ in a region where MLKAPS did not perform well. MLKAPS picks an average solution, while the MKL chooses among the best available. In that particular case, it was due to an incorrect prediction from the surrogate model, which is solved by increasing the number of samples until the model has converged, like shown in the previous section with figure 8.

The second Histogram 9 (c) shows the distribution of performance for $(n = 4500; m = 1600)$ in the region where MLKAPS far outperforms the MKL. In this region, MLKAPS selects a good solution as expected, while Intel MKL uses a surprisingly inefficient tuning. This corresponds to a blind spot in Intel MKL tuning strategy. We replicated this behavior on the same region on a Cascade Lake processor, but it is absent from the Saphire Rapids Processor. Overall, with only 7,000 samples on the KNM, MLKAPS matches or outperforms the MKL reference tuning on 74% of the input space with a geomean speedup of 20%, and enabled us to identify a critical blind spot.

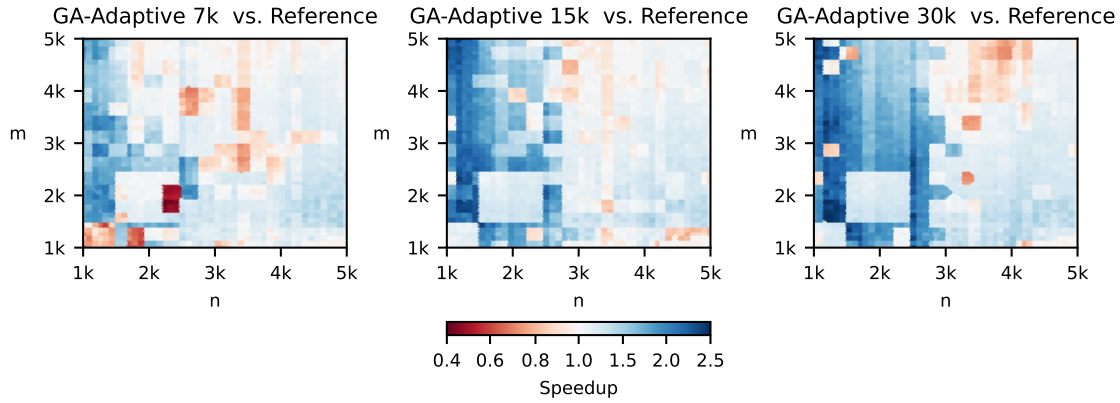

Fig. 10. **(SPR): Speedup of MLKAPS's decision tree compared to the reference Intel MKL configuration on `dgetrf` (LU)** for 7K, 15K, and 30K samples. 46x46 validation grid. The MLKAPS model performs better with more samples, with a geometric mean ×1.3 speedup. The blocked patterns (Light-blue square region in the left-most plot) are due to the decision trees' partition strategy.

5.3.2 Results on the SPR. We repeated these experiments on the SPR for 7k, 15k, and 30k samples to show how MLKAPS performs when increasing the number of samples, as well as to highlight behavioral differences due to the hardware.

Figure 10 shows that the objective space is significantly different from the KNM processor and that MLKAPS can still find a significant speedup in modern architectures. Similarly to the KNM experiments, there are clear regions of improvement and some regressions. As the knowledge of the model increases with the number of samples, we can observe that at 30K samples, almost no significant regression remains. With 30k samples, MLKAPS has a mean speedup of 30%. In addition to this mean speedup, we further distinguished the percentage of "regressions" points with a slowdown, and "progressions" with a speedup. For 30k samples, we observe 15% regressions with a mean slowdown of ×0.10, and 85% of progressions with a mean speedup of ×1.38.

5.4 Comparison with other auto-tuners

5.4.1 Comparison to Optuna. To show the benefits of transfer learning using MLKAPS' surrogate model, we compare our approach with Optuna [1], a renowned hyperparameter autotuner, on the Intel MKL `dgeqrf` (QR) kernel. Optuna does not have a global model of the objective space, and the points are optimized individually.

This kernel shares the same two inputs and eight design parameters as Intel MKL `dgetrf` (LU). We compare both tools on a 46×46 regular validation grid with 30k samples.

In Figure 11 we observe that MLKAPS has a geomean speedup of 18% on this kernel, improving the reference configuration on 85% of the input space. The mean slowdown on regressions is 29%, the speedup on progressions 31%. This kernel has a better baseline configuration than `dgetrf`. Furthermore, MLKAPS outperforms Optuna on 98% of the input space, with a geomean speedup of 36%.

These experiments show the benefits of using a surrogate model for transfer learning. However, we found that Optuna's ease of use and generalization capabilities were useful to tune MLKAPS algorithms, including the model hyperparameters to some extent. When tuning kernels for a single input, Optuna is competitive, but MLKAPS is more efficient when targeting large input spaces.

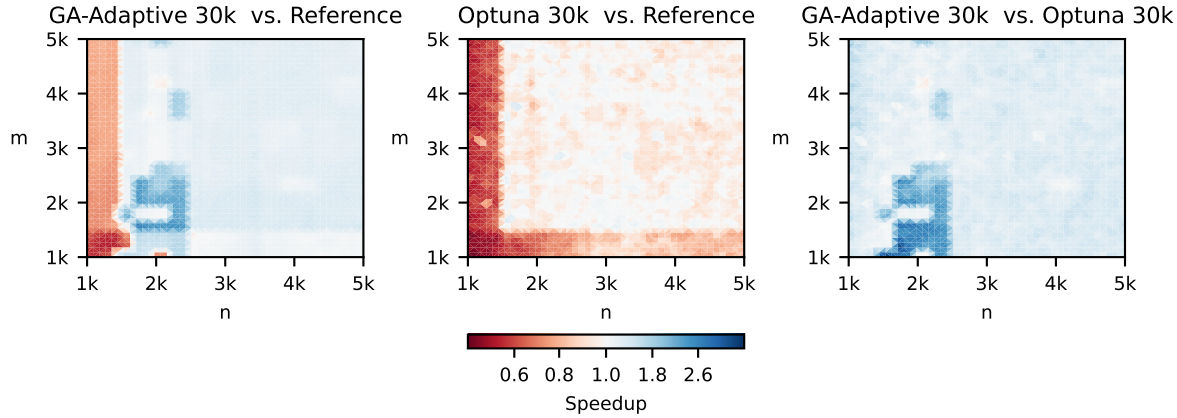

Fig. 11. **(SPR): Speedup map of MLKAPS vs. Optuna on the MKL `dgeqrf` kernel.** MLKAPS has a geomean 18% speedup over the MKL, but shows regions with significant regressions. In these regions, the MKL is near-optimal and very difficult to outperform.

5.4.2 Expert knowledge injection. The regressions observed in figure 11 are due to the MKL embedding an optimal tuning that is very hard to find using statistical sampling. In an industrial context, such regressions are not acceptable despite the achievement of speedups in other regions. It is worth noticing that each region being independent, it is possible to combine MLKAPS speedups with the MKL one in a single configuration tree to get the best of both worlds. This can be done by selecting the best solution on each input between the MKL reference and the MLKAPS optimized candidate. Such combinations ensure that we always select the best-known configuration, either from auto-tuning exploration or from previous expert knowledge. This method can also be applied with multiple MLKAPS runs to progressively refine the decision trees. Figure 12 shows the benefits of using this method: Expert trees outperform MKL

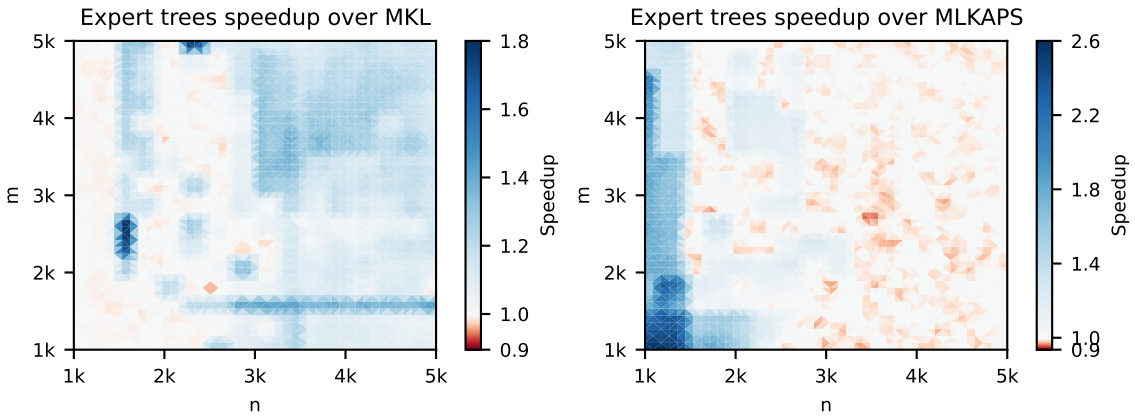

Fig. 12. **(SPR): Expert tree combining MKL expert knowledge with MLKAPS auto-tuning solutions compared to MKL and MLKAPS decision trees.** MLKAPS 15k run. We eliminate all regressions (points remaining under 1.0 speedup are due to measurement noise) and bring significant improvements to the reference MKL configuration. The geometric mean speedup over the MKL is 11%.

for all inputs with a geometric mean speedup of 11%, and all regressions are removed. A 15k sample MLKAPS run was used to train the expert trees.

5.4.3 Comparison to GPTune. Bayesian optimization is a popular technique for auto-tuning hyperparameters in HPC [21, 23]. GPTune [20] is a state-of-the-art black-box autotuning framework that leverages a sophisticated Bayesian optimization framework. It builds a set of Gaussian processes and combines them using a linear model of coregionalization (LMC) to perform transfer learning and efficiently optimize multiple inputs at once.

GPTune and MLKAPS have different strategies for learning in a large design space. MLKAPS decouples the sampling and optimization phases by first building a surrogate model of the entire space. Once the surrogate model is trained, MLKAPS can optimize for any inputs. GPTune does not do this decoupling: The user must select the set of inputs for which GPTune will provide tuned design parameters, which also limits which inputs will be sampled. It can also perform random sampling for automated input selection.

The advantage of GPTune's approach is that the outputs are always validated against a directly measured sample; whereas MLKAPS' choices rely on the quality of the surrogate model. Recent versions of GPTune include the two-level transfer learning algorithm (TLA2), which allows the extrapolation of solutions on points that have not been explored, much like the MLKAPS decision tree(s). However, GPTune extrapolation capabilities suffer from being constrained to sampling a limited set of inputs, therefore, completely missing performance cliffs, overfit to special cases, or miss transitions in the objective space. This problem is mitigated in MLKAPS by using a surrogate model that does not restrict the input space as explained in Section 4.2. Compensating for this would require GPTune to increase the number of tasks, which we will see in a later section, does not scale. Furthermore, for predicting configurations at runtime in a production environment, they would require their tool to be integrated with the application, while MLKAPS provides simple decision trees that can be embedded in the kernel.

We used the latest docker image on GPTune's repository (`liuyangzhuan/gptune:4.5`) with the LMC (Linear Coregionalization Model). Minor modifications were required to port the TLAII algorithm from the latest Github version, as it is not available in the Docker image. We compared both tools on two linear algebra kernels: the ScaLAPACK PDGEQRF (QR) described in their paper and the Intel MKL `dgetrf` (LU).

ScaLAPACK PDGEQRF. First, we ran MLKAPS on the ScaLAPACK `pdgeqrf` experiment described in the GPTune original paper [20]. This experiment was designed by the GPTune authors and optimizes a QR factorization kernel. They ran this experiment on up to 64 KNM nodes on the Cori cluster while we only have one KNM node available, which might explain differences in our results. In their paper, they optimized up to 10 input tasks with a total budget of 100 samples, which is significantly less than the usual budget required for MLKAPS experiments.

The current version of MLKAPS does not handle constrained optimization, so we reformulated the problem as a set of free parameters. The problem parameters are listed in Table 1. Original constraints are expressed as a set of inequalities (i.e., $mb \times p \times 8 \leq m$) To satisfy the inequalities, we restricted three variables, mb, nb, and n_node, and computed their upper and lower bounds according to the remaining variables. After this process, the admissible values for each bound variable lay in a convex interval, so we can replace the bound variable with a free parameter in $[0, 1]$ that selects a point on the interval by linear interpolation (i.e. the previous example becomes $1 \leq mb \leq \frac{m}{8p}$). We ran GPTune on this transformed space to validate our approach and found that GPTune had a harder time fitting on these new variables. Note that only MLKAPS used this reformulation and that GPTune ran on the original problem.

We choose 64 points chosen as a regular 8×8 grid over matrix sizes $3072 \leq n, m \leq 8072$ as inputs for GPTune to fit their scalability limitation as described in a later experiment.

Name	Description	Reformulation
(m, n)	Matrix size in the (1^{st},2^{nd}) axis	Identical
p	Process grid size along the 1^{st} axis	Identical
$mb \rightarrow \alpha$	Block size along m	$mb = lerp(\alpha, 1, min(\frac{m}{8p}, 16))$
$npernode \rightarrow \beta$	Nb. of process per compute node	$npernode = p + lerp(\beta, 0, 30 - p)$
$nb \rightarrow \gamma$	Block size along n (np is the total number of processors, a constant)	$nb = lerp(\gamma, 1, min(\frac{np}{8npernode}, 16))$

Table 1. **Parameters of PDGEQRF and their respective reformulation.** We note $lerp(\alpha, lb, ub)$ the linear interpolation between lb and ub with $\alpha \in [0, 1]$

We run GPTune and MLKAPS with an increasing number of samples up to 1024 and record the tuning cost and the performance achieved in figure 13. Both tools converge to a mean execution time of 2.09s (2.093 for GPTune and 2.096 for MLKAPS). Nevertheless, MLKAPS converges much faster than GPTune towards the optimal configuration: MLKAPS reaches optimal configurations with less than 200 samples, while GPTune requires 500 samples. We found that the objective in this experiment is almost entirely dominated by the parameter p, which explains why such a low number of samples is required for convergence.

Moreover, MLKAPS has a lower tuning cost than GPTune for all sample counts and is up to 2.44× faster for 1024 samples. This leads to one of the main pitfalls of GPTune's approach compared to MLKAPS: its optimization pipeline does not scale to a large number of samples and grid points.

MKL `dgetrf`. To confirm GPTune scalability issues, we decided to run the tool on Intel `dgetrf` (LU) kernel presented earlier in Section 5.3.2. We previously saw that MLKAPS continuously improved with an increasing number of samples up to 30k, leading us to believe that this experiment is significantly harder than ScaLAPACK `pdgeqrf` as it requires more samples.

For this experiment, we run GPTune with 8 random input tasks and then use TLAII for predicting on a 46×46 grid. The run crashed due to an out-of-memory issue, as visible in figure 14 which represents memory usage according to the number of samples. We found that GPTune's approach suffers from a scalability issue, both in memory and execution

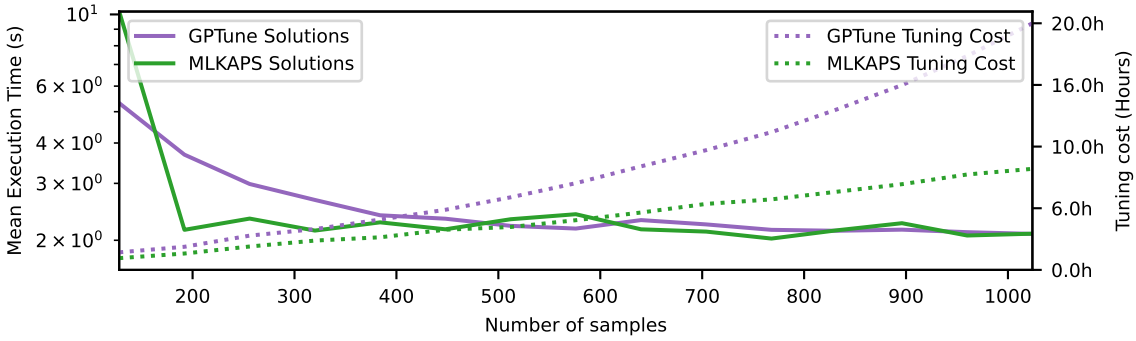

Fig. 13. **(KNM): GPTune vs. MLKAPS on ScaLAPACK PDGEQRF (QR).** Both GPTune and MLKAPS converge to an equivalent optimized solution. Nevertheless, MLKAPS requires less tuning time and collected samples to find a near-optimal configuration.

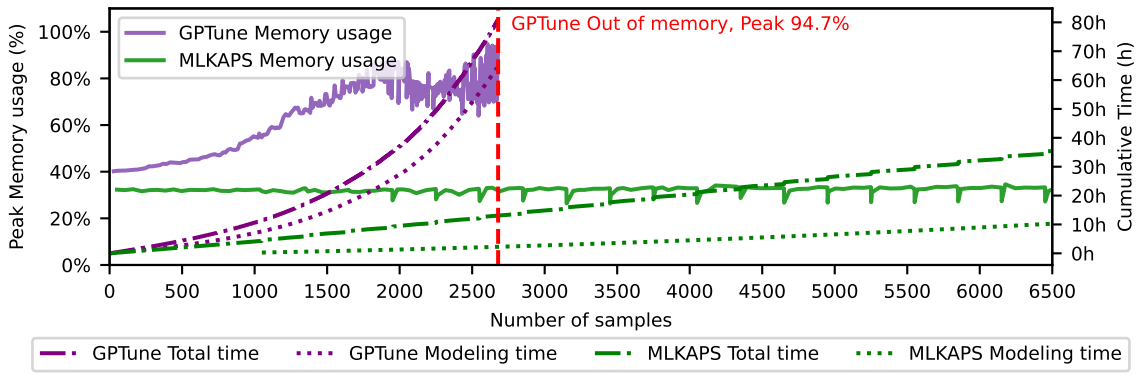

Fig. 14. **(KNM): Evolution of peak memory usage and sampling time on the `dgetrf` (LU) experiment, with 16 tasks and a 7k sampling budget.** GPTune memory and modeling time increase non-linearly with the number of samples. After 2512 samples, GPTune was killed by the operating system, having consumed all available memory. The instability in the GPTune memory usage after 1500 samples is likely due to the operating system swapping memory regions to disk. MLKAPS exhibits linear scaling in time and constant memory usage, handling a large number of samples well.

time, and is dependent on the number of samples and the number of tasks. From our understanding, this issue comes from the covariance matrix needed by LCM of size $\epsilon\delta \times \epsilon\delta$ (Sec. 3.3 in [20]) with ϵ the number of samples per task, and δ the number of tasks. MLKAPS scales linearly in memory and time with the number of samples, and most of the runtime is spent collecting samples. Overall, MLKAPS achieves a 15% speedup on `dgetrf` when compared to GPTune on the KNM and finishes in half the time with $\times 3$ the number of samples. Overall, we believe that both MLKAPS and GPTune approaches are valid, but target different scopes:

- MLKAPS is most useful for building decision trees for predicting configurations on a large range of inputs, in cases where thousands of measures are feasible.
- On the opposite side, GPTune provides a sophisticated optimization framework that has the advantage of handling constraints over the parameter space and is well suited to optimize thoroughly over a few well-chosen inputs but cannot scale to larger experiments.

6 DISCUSSIONS AND FUTURE WORKS

Our design already provides very encouraging results. Future works to improve MLKAPS include:

- Extend our use-cases to more kernels and hardware architectures.
- Handle constraint without explicit reformulation from the user.
- Improve the decision trees, measure both overhead, interpretability, and performance. Currently, MLKAPS uses one tree per design parameter, and we are considering merging into a single and using more complex splitting criteria.
- Dynamic Optimization Grid: Currently, MLKAPS uses a fixed regular grid to optimize on. However, this method was shown to miss local optima in some cases and is inefficient as neighboring points in the grid often share similar configurations, which in turn induces useless splits in the tree(s).

- Extending support for expert-knowledge injection to leverage existing tunings in cases similar to `dgetrf`, or allow user to inject better solution in MLKAPS model enabling to re-tune the solution provided by the black-box model approach.
- Introducing a metric for the surrogate model that measures its optimization capabilities.
- Our preliminary works show that Optuna [1] may be used to tune MLKAPS hyper-parameters, relieving the non-expert end-user from complex understanding and model tuning process.

7 CONCLUSION

This article shows how manual tuning is error-prone due to human bias and often relies on inefficient optimization methods. We introduced MLKAPS, an auto-tuning tool that generates decision trees that can be embedded into HPC kernels. We described the implementation of GA-Adaptive, an optima-biased sampler that outperformed the other sampling strategies.

We applied MLKAPS to an industrial parallelized kernel taken from Intel MKL and compared how the different sampling techniques led to different surrogates and tuning performances. GA-Adaptive offers better tuning results than other techniques and improves the surrogate's local accuracy. We compared the tuning solutions proposed by MLKAPS with the reference hand-tuned Intel MKL on the `dgetrf` kernel. We showed that MLKAPS matches or outperforms Intel `MKL` on 85% of the input space with an average speedup of 38%.

We compared MLKAPS to the state-of-the-art auto-tuner GPTune and Optuna on kernels from both ScaLAPACK and Intel MKL. MLKAPS is significantly more scalable and outperforms GPTune on problems with large design space. MLKAPS global GBDT model allows it to outperform Optuna for the same number of samples.

This article highlighted the advantages of an auto-tuning tool capable of generating decision trees for dynamic runtime tuning. MLKAPS is open-source and available at https://github.com/MLCGO/MLKAPS.

This project received funding from an Intel Higher Education grant.

REFERENCES

[1] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. 2019. Optuna: A Next-generation Hyperparameter Optimization Framework. In *Proc. of the 25th ACM SIGKDD Int. Conference on Knowledge Discovery and Data Mining*.

[2] Andreea Anghel, Nikolaos Papandreou, Thomas Parnell, Alessandro De Palma, and Haralampos Pozidis. 2018. Benchmarking and Optimization of Gradient Boosting Decision Tree Algorithms. *arXiv e-prints*, Article arXiv:1809.04559 (Sept. 2018), arXiv:1809.04559 pages. arXiv:1809.04559 [cs.LG]

[3] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. 2002. Finite-time Analysis of the Multiarmed Bandit Problem. *Machine Learning* 47 (05 2002), 235–256. https://doi.org/10.1023/A:1013689704352

[4] Nathan Beckmann and Daniel Sanchez. 2015. Talus: A simple way to remove cliffs in cache performance. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. 64–75.

[5] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. 2011. Algorithms for Hyper-Parameter Optimization. In *Advances in Neural Information Processing Systems*, J. Shawe-Taylor, R. Zemel, P. Bartlett, F. Pereira, and K.Q. Weinberger (Eds.), Vol. 24. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2011/file/86e8f7ab32cfd12577bc2619bc635690-Paper.pdf

[6] James Bergstra, Nicolas Pinto, and David Cox. 2012. Machine learning for predictive auto-tuning with boosted regression trees. In *2012 Innovative Parallel Computing*. 1–9.

[7] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. 1997. *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA.

[8] J. Blank and K. Deb. 2020. pymoo: Multi-Objective Optimization in Python. *IEEE Access* 8 (2020), 89497–89509.

[9] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. Learning to optimize tensor programs. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems* (Montréal, Canada) *(NIPS'18)*. Curran Associates Inc., Red Hook, NY, USA, 3393–3404.

[10] Matthias Christen, Olaf Schenk, and Helmar Burkhart. 2011. PATUS: A Code Generation and Autotuning Framework for Parallel Iterative Stencil Computations on Modern Microarchitectures. In *2011 IEEE International Parallel & Distributed Processing Symposium*. 676–687.

[11] Pablo de Oliveira Castro, Eric Petit, Jean Christophe Beyler, and William Jalby. 2012. ASK: Adaptive Sampling Kit for Performance Characterization. In *Euro-Par 2012 Parallel Processing*, Vol. 7484. Springer, Greece, 89–101.

[12] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation* 6, 2 (2002), 182–197.

[13] Matteo Frigo. 1999. A Fast Fourier Transform Compiler. *SIGPLAN Not.* 34, 5 (1999), 169–180.

[14] Davide Gadioli, Ricardo Nobre, Pedro Pinto, Emanuele Vitali, Amir H. Ashouri, Gianluca Palermo, Joao Cardoso, and Cristina Silvano. 2018. SOCRATES — A seamless online compiler and system runtime autotuning framework for energy-aware applications. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 1143–1146.

[15] Nikolaus Hansen. 2023. The CMA Evolution Strategy: A Tutorial. arXiv:1604.00772 [cs.LG] https://arxiv.org/abs/1604.00772

[16] Intel. 2024. Intel Math Kernel Library Documentation. https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl-documentation.html

[17] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. 2017. Lightgbm: A highly efficient gradient boosting decision tree. *Advances in neural information processing systems* 30 (2017), 3146–3154.

[18] Sotiris Kotsiantis. 2013. Decision trees: A recent overview. *Artificial Intelligence Review* (04 2013), 1–23.

[19] Chendi Li, Yufan Xu, Sina Mahdipour Saravani, and Ponnuswamy Sadayappan. 2024. Accelerated Auto-Tuning of GPU Kernels for Tensor Computations. In *Proceedings of the 38th ACM International Conference on Supercomputing* (Kyoto, Japan) *(ICS '24)*. Association for Computing Machinery, New York, NY, USA, 549–561. https://doi.org/10.1145/3650200.3656626

[20] Yang Liu, Wissam M Sid-Lakhdar, Osni Marques, Xinran Zhu, Chang Meng, James W Demmel, and Xiaoye S Li. 2021. GPTune: multitask learning for autotuning exascale applications. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 234–246.

[21] Piotr Luszczek, Wissam M Sid-Lakhdar, and Jack Dongarra. 2023. Combining multitask and transfer learning with deep Gaussian processes for autotuning-based performance engineering. *The International Journal of High Performance Computing Applications* (2023), 10943420231166365.

[22] M. D. McKay, R. J. Beckman, and W. J. Conover. 1979. A Comparison of Three Methods for Selecting Values of Input Variables in the Analysis of Output from a Computer Code. *Technometrics* 21, 2 (1979), 239–245.

[23] Harshitha Menon, Abhinav Bhatele, and Todd Gamblin. 2020. Auto-tuning Parameter Choices in HPC Applications using Bayesian Optimization. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 831–840. https://doi.org/10.1109/IPDPS47924.2020.00090

[24] Dheya Mustafa. 2022. A Survey of Performance Tuning Techniques and Tools for Parallel Applications. *IEEE Access* 10 (2022), 15036–15055.

[25] OpenMP Architecture Review Board. 2021. OpenMP Application Program Interface Version 5.2. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf

[26] M. Puschel, J.M.F. Moura, J.R. Johnson, D. Padua, M.M. Veloso, B.W. Singer, Jianxin Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R.W. Johnson, and N. Rizzolo. 2005. SPIRAL: Code Generation for DSP Transforms. *Proc. of the IEEE* 93, 2 (2005), 232–275.

[27] R.A. Rutenbar. 1989. Simulated annealing algorithms: an overview. *IEEE Circuits and Devices Magazine* 5, 1 (1989), 19–26. https://doi.org/10.1109/101.17235

[28] Philippe Tillet and David Cox. 2017. Input-Aware Auto-Tuning of Compute-Bound HPC Kernels. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '17)*. Association for Computing Machinery, New York, NY, USA, Article 43, 12 pages.

[29] Matej Črepinšek, Shih-Hsi Liu, and Marjan Mernik. 2013. Exploration and Exploitation in Evolutionary Algorithms: A Survey. *ACM Comput. Surv.* 45, 3, Article 35 (jul 2013), 33 pages.

[30] Michel Verleysen and Damien François. 2005. The Curse of Dimensionality in Data Mining and Time Series Prediction. In *Computational Intelligence and Bioinspired Systems*, Joan Cabestany, Alberto Prieto, and Francisco Sandoval (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 758–770.

[31] R. Clint Whaley. 2011. *ATLAS (Automatically Tuned Linear Algebra Software)*. Springer US, Boston, MA, 95–101.

[32] Jiarong Xing, Leyuan Wang, Shang Zhang, Jack Chen, Ang Chen, and Yibo Zhu. 2021. Bolt: Bridging the Gap between Auto-tuners and Hardware-native Performance. *CoRR* abs/2110.15238 (2021). arXiv:2110.15238 https://arxiv.org/abs/2110.15238

[33] Yifan Zhao, Hashim Sharif, Vikram Adve, and Sasa Misailovic. 2024. Felix: Optimizing Tensor Programs with Gradient Descent. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 367–381. https://doi.org/10.1145/3620666.3651348

[34] Lianmin Zheng, Ruochen Liu, Junru Shao, Tianqi Chen, Joseph E. Gonzalez, Ion Stoica, and Ameer Haj Ali. 2021. TenSet: A Large-scale Program Performance Dataset for Learned Tensor Compilers. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*. https://openreview.net/forum?id=aIfp8kLuvc9