Future Generation Computer Systems 179 (2026) 108358

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Automatic tuning based on hardware performance counters and machine learning

Suren Harutyunyan Gevorgyan[a,*], Eduardo César[a], Anna Sikora[a], Jiří Filipovič[b], Jordi Alcaraz[c]

[a] Universitat Autònoma de Barcelona, DACSO, Bellaterra, 08193, Spain
[b] Masaryk University, ICS, Brno-Královo Pole, 602 00, Czech Republic
[c] University of Oregon, OACISS, Eugene, 97403, USA

ARTICLE INFO

Keywords:
Hardware performance counters
Automatic dimension reduction
Machine learning ensembles
Tuning parameter optimization
Parallel region classification

ABSTRACT

This paper presents a Machine Learning (ML) methodology for automatically tuning parallel applications in heterogeneous High Performance Computing (HPC) environments using Hardware Performance Counters (HwPCs). The methodology addresses three critical challenges: counter quantity versus accessibility tradeoff, data interpretation complexity, and dynamic optimization needs. The introduced ensemble-based methodology automatically identifies minimal yet informative HwPC sets for code region identification and tuning parameter optimization. Experimental validation demonstrates high accuracy in predicting optimal thread allocation (>0.90 K-fold accuracy) and thread affinity (>0.95 accuracy) while requiring only 4–6 HwPCs. Compared to search-based methods like OpenTuner, the methodology achieves competitive performance with dramatically reduced optimization time. The architecture-agnostic design enables consistent performance across CPU and GPU platforms. These results establish a foundation for efficient, portable, automatic, and scalable tuning of parallel applications.

1. Introduction

The tuning of parallel applications in modern High Performance Computing (HPC) systems presents significant challenges due to increasing architectural complexity and heterogeneity. Hardware Performance Counters (HwPCs) provide detailed low-level performance data offering valuable insights into application behavior, leading to the hypothesis that HwPCs and Machine Learning (ML) methodologies can be used to optimize parallel code regions, defined as distinct computational blocks within an application that can be independently optimized. However, their effective use presents several challenges:

1. **Monitoring Limitations**: Hardware limitations prevent simultaneous access to all HwPCs. Measuring fewer, carefully selected events is crucial for accuracy, allowing for more measurement time per event when using techniques such as multiplexing.
2. **Scalability and Data Volume Constraints**: The volume of HwPC data, especially from GPUs, with hundreds of associated counters, implies serious scalability challenges. Manual analysis becomes impractical, necessitating automated approaches.
3. **Runtime Decision Requirements**: HPC environments require rapid analysis. This demands efficient performance models that use a minimal set of HwPCs that accurately represent application behavior to enable runtime optimization decisions.

These challenges have spurred interest in leveraging ML for HwPC-based tuning. Our previous research in [1,2] demonstrated promising results in identifying OpenMP code regions and predicting the optimal number of threads using a reduced set of HwPCs. However, this initial approach had critical limitations: feature selection relied on manual, analytical methods (PCA and Correlation) that did not scale to richer HwPC architectures, and used the same HwPC set for both region identification and parameter tuning, a conceptual flaw, as the optimal counters for these tasks likely differ.

This caused interpretability issues due to manual HwPC selection, limited portability across architectures, and inconsistent prediction accuracy using a the same set of HwPCs for classification and tuning.

In [3], we proposed a fully ML-based automated methodology to select the minimum number of HwPCs necessary to identify an OpenMP code region, addressing the manual selection flaw. However, a critical gap remained: the lack of an end-to-end automated methodology that seamlessly identifies parallel code regions and optimizes Tuning Parameters (TPs) using different, task-specific HwPC sets across diverse architectures.

* Corresponding author.
E-mail addresses: Suren.Harutyunyan@uab.cat (S. Harutyunyan Gevorgyan), Eduardo.Cesar@uab.cat (E. César), Anna.Sikora@uab.cat (A. Sikora), fila@ics.muni.cz (J. Filipovič), jordia@uoregon.edu (J. Alcaraz).

https://doi.org/10.1016/j.future.2025.108358
Received 31 July 2025; Received in revised form 16 December 2025; Accepted 29 December 2025
Available online 30 December 2025

To bridge this gap, an effective automatic tuning system must provide:

- **Efficient HwPC Selection**: Identifying the most informative subset of HwPCs to reduce overhead.
- **Automated Pattern Recognition**: Detecting patterns for accurate code region characterization.
- **Predictive Tuning**: Learning optimal TP configurations from performance data.
- **Cross-Architecture Portability**: Adapting to diverse hardware like CPUs and GPUs.

The primary contributions of this paper, which address the aforementioned gap and limitations, are:

- **A Novel End-to-End ML Methodology**: We introduce a new approach that builds upon [3] by not only automating the selection of a reduced HwPC set but also determining distinct, task-specific sets: one for code region identification and separate sets for optimizing each region's associated TPs.
- **A Portable Multi-Architecture Solution**: We extend this methodology to be effective across different architectures (CPU and GPU), ensuring consistent performance without manual, architecture-specific modifications.

The remainder of this paper is organized as follows. Section 2 shows the background, including previous works on HwPC reduction and technical concepts. Section 3 describes the novel and automatic ensemble methodology for automatic HwPC reduction, code region identification, and TP optimization. Section 4 shows the evaluation of the proposed methodology on different architectures. Section 5 presents related work. Finally, Section 6 concludes the paper.

2. Background

This section discusses the integration of HwPCs and ML methodologies for parallel computing performance optimization, reviewing prior research on HwPC-based code region identification and ML-driven TP optimization, their achievements and limitations. The section also introduces the automatic ML ensemble methodology and constituent algorithms, as a framework for accurate TP optimization, along with evaluation metrics to assess the predictive capabilities.

2.1. Previous work

In [1] the authors reduced HwPC numbers for code region identification using PCA (Principal Component Analysis) and Linear Correlation Analysis (LCA).

PCA is a dimensionality reduction technique that projects data into a new coordinate system that emphasizes variability patterns while eliminating less informative dimensions, thereby facilitating data exploration and analysis. It was employed to assess the visual separability of data classes.

LCA is a statistical method that quantifies the linear relationship between two continuous numerical variables, producing a correlation coefficient ranging from -1 to 1. When variables exhibit perfect correlation (coefficient of 1 or -1), one variable's value can be predicted from the other through appropriate linear transformation, while a coefficient of 0 indicates no linear relationship exists. The analysis is used to reduce HwPCs, thus variables exhibiting high correlation coefficients are removed, after which PCA is applied again to confirm that the dimensional reduction maintains adequate characterization of the parallel code region. Using this approach the authors successfully reduced the number of HwPCs from 58 to 20.

A related study [2] focused on optimizing the number of threads through ML methodologies. The authors evaluated multiple ML techniques including Logistic Regression, Artificial Neural Network (ANN), and Decision Trees (DT), ultimately retaining only the ANN and DT models due to their significantly superior accuracy compared to other approaches. This was validated with the STREAM [4] and PolyBench [5] benchmarks.

Both studies exhibited limitations. The first study's reliance on manual selection for HwPC reduction introduced overhead and compromised the scalability of the approach. The second study revealed a critical limitation in the model's predictive accuracy. Specifically, for Stride code regions, the model consistently underestimated the optimal number of threads, leading to significant performance degradation. This underestimation, to a lesser extent, was also observed in several other code regions, indicating a general weakness rather than an isolated issue.

In [3], the first study's limitation was addressed with an automatic HwPC reduction methodology, which was able to accurately identify code regions across different architectures. The methodology employs an ML ensemble to identify minimal HwPC sets necessary for effective code region identification. While this approach effectively reduced HwPC sets for code region identification, it did not address the TP optimization weakness.

The present work further advances the methodology by employing ML ensembles not only to reduce the HwPC sets for both, code region identification and TP optimization, but also to train dedicated ensembles on these minimal sets. These ensembles are then used for code region identification and TP optimization, providing a unified and automated framework that extends prior approaches. Therefore, this research addresses both previous limitations through an automatic bifurcated HwPC reduction strategy that leverages distinct HwPC sets for code region identification and TP optimization, thus enhancing discriminative capability and predictive accuracy. Moreover, we have extended the methodology to support CPU and GPU architectures.

Thus, while building on prior work, the present contribution represents a substantial step from code region identification toward a complete automatic optimization methodology.

2.2. Dataset construction

ML methodologies excel at extracting insights from complex datasets in computational performance analysis. The quality of the model's predictions depends on comprehensive data-collection, requiring sophisticated approaches to data acquisition and preprocessing.

In [6] an approach for systematically building balanced datasets of Hardware Performance Counters (HwPCs) for OpenMP parallel regions was introduced. This method takes into account all possible combinations of architectural characteristics (e.g., number of cores, memory hierarchy), region characteristics (e.g., data layout and size), compiler optimizations, and parallelization strategies (e.g., number of threads, affinity). Consequently, characterizing each code region requires a significant number of executions. The approach utilizes the Performance Application Programming Interface (PAPI) [7,8] to collect only preset HwPCs, ensuring cross-architecture generalizability. To overcome the limitation on the number of HwPCs that can be monitored concurrently, the execution of each region configuration is repeated for each group of compatible events; the resulting measures are then concatenated to create a single, comprehensive characterization of the region.

Furthermore, the methodology requires that problem sizes be directly proportional to the memory size at each level of the memory hierarchy. Specifically, for on-processor caches (L1, L2, L3), problem sizes must be proportional to the number of physical cores per processor, while for memories outside the processor, they must be proportional to the number of processors in the system. For each private cache level, problem sizes are defined starting with the size of one private cache and multiplied by the different core configurations, ending with the accumulated size of the private caches in the same level. For each shared cache level, the problem sizes are bigger than the accumulated size of the lower level cache and slightly lower than the maximum shared memory in the current cache level. Finally, for the main memory, the initial

Table 1
Summary of the execution parameters for a specific platform (Xeon E5-4620).

Parameter		Values for Xeon E5-4620
Counter Sets	S	12
Threads	T	32
Problem Sizes	P	29
OpenMP TPs	OP	11
Repetitions	N	100
Total	E	**12249600**

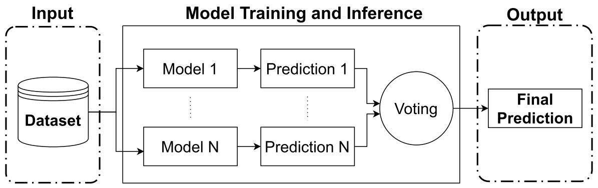


Fig. 1. Theoretical model ensembling process.

problem size is bigger than the last level cache of the processor, and the other sizes are obtained gradually increasing the necessary memory. The last problem size is 1.5 times the aggregated size of the last level cache of all the processors in the system.

The methodology demonstrates robustness through systematic variation of OpenMP parameters, manipulating dimensions including number of threads, thread affinity policies, scheduling strategies, and chunk sizes. The execution strategy involves multiple repetitions across different configurations that can be calculated as:

$$E = S \times T \times P \times OP \times N, \quad (1)$$

where E denotes executions, S HwPC sets, T number of threads, P problem sizes, OP OpenMP TP combinations, and N repetitions.

The final dataset undergoes preprocessing, where the HwPC values are normalized, and null values and zero-variance features are removed. Table 1 presents a summary of the execution parameters associated with their particular values for the Xeon E5-4620 platform. From this dataset, optimal TP values are identified as configurations minimizing the objective function (e.g. $P_i(X)$ or execution time) for the number of threads, affinity, or scheduling/chunk-size. These optimal configurations serve as ground truth for training the ensemble models.

2.3. Ensemble methodology

Ensemble modeling combines multiple individual models (weak or base learners) to create a robust and accurate prediction system [9,10]. As shown in Fig. 1, this process comprises several key components:

Model Diversity: The ensemble consists of multiple independent models working in parallel. Each weak model provides a unique perspective using different algorithms or training parameters, capturing various aspects of the underlying patterns while reducing individual model biases.

Input Processing: Test input is simultaneously fed into all models, allowing each to analyze the same data independently and generate predictions based on unique characteristics and learned patterns.

Model Integration through Voting: Individual predictions are combined through **majority voting** (most frequent prediction), **weighted voting** (based on model confidence), or **soft voting** (aggregating probability distributions), producing more stable and accurate predictions than any single model.

Performance Benefits: Ensemble methods demonstrate superior performance by **averaging out** different error types, **reducing sensitivity** to outliers and noise, and improving **generalization** across diverse input data.

2.4. Ensemble algorithms

The ensemble models were selected to maximize complementarity across algorithmic paradigms specifically for HwPC feature spaces. Four models were selected from nine initial candidates through evaluation based on computational efficiency during training and predictive performance metrics. Models with excessive training requirements or inferior accuracy, precision, and recall were discarded to ensure practical deployment in resource-constrained environments.

Logistic Regression with Elastic Net [11–14] addresses the multicollinearity inherent in HwPC data (e.g., correlated cache miss events) while providing feature selection through L1 regularization. This is crucial because most HwPCs measure related architectural events. It excels when linear relationships exist between HwPC combinations and performance outcomes, particularly for compute-bound code regions where instruction counts linearly predict execution time.

Random Forest [15] addresses nonlinear interactions in memory-intensive code regions, where relationships between cache misses, memory bandwidth, and performance are interdependent. Its resilience to outliers is essential given that HwPC values can span several orders of magnitude. Its feature importance mechanism also provides interpretability for identifying which architectural events most influence performance.

XGBoost [16] addresses class imbalances that may arise in the dataset (where optimal configurations may be rare) through its weighting mechanisms. Its boosting framework captures dependencies between HwPC events and provides strong performance in high-dimensional, sparse feature spaces.

TabNet [17] handles the heterogeneous nature of HwPC data (mixing instruction counts, temporal cycles, and throughput rates) through its attention mechanism, automatically learning which HwPC groups are relevant for specific code regions. Unlike other models, it performs simultaneous feature selection and prediction, which is crucial when dealing with 50+ HwPCs where many may be irrelevant.

This diversity enables handling different aspects of the HwPC feature space: from linear to complex nonlinear relationships, from balanced to imbalanced classes, and from simple to hierarchical decision boundaries.

2.5. Ensemble evaluation metrics

Evaluation of model performance is critical for assessing the efficacy and reliability of ML methodologies in performance optimization. Five key indicators were selected based on the requirements of tuning tasks: accuracy, precision, recall, F1-Score, and ROC AUC. Each metric addresses distinct challenges in performance optimization.

Accuracy quantifies overall correctness but may be misleading in imbalanced scenarios (e.g., when the optimal configuration occurs in only 15% of cases). Nevertheless, it provides a baseline measure of general performance. To obtain a more reliable estimate of generalization, we adopted K-fold cross-validation with $k = 5$ balancing statistical robustness and computational feasibility. Cross-validation accuracy is defined as the mean accuracy across k validation folds, offering an estimate that is less sensitive to data partitioning.

Precision is relevant because false positives are costly. Recommending a suboptimal configuration can waste significant runtime. High precision ensures that when the model recommends a configuration, it is indeed the optimal or near-optimal configuration.

Recall addresses the complementary risk of missing optimal configurations. In tuning scenarios, low recall means the model overlooks optimal configurations, leaving potential performance gains undiscovered.

F1-Score balances precision and recall, which is essential because neither metric alone suffices. A model with high precision but low recall might only predict in obvious cases (e.g., always recommending maximum threads) while missing subtler optimal configurations. The

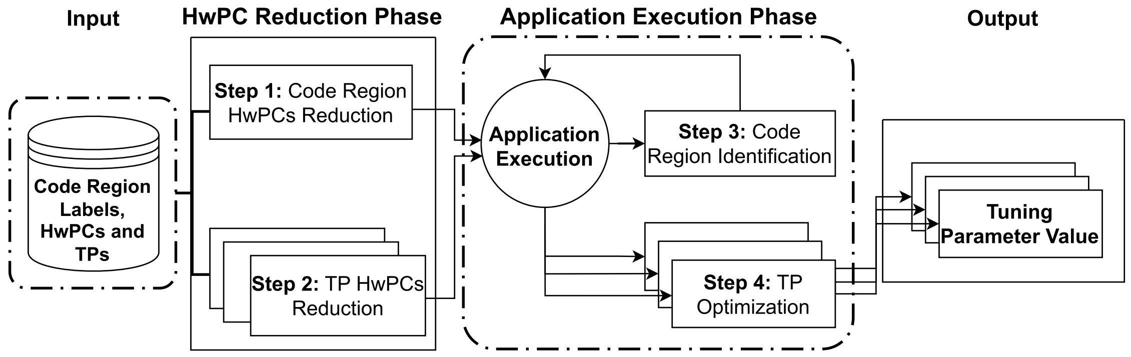

Input
HwPC Reduction Phase
Application Execution Phase
Output
Code Region Labels, HwPCs and TPs
Step 1: Code Region HwPCs Reduction
Step 2: TP HwPCs Reduction
Application Execution
Step 3: Code Region Identification
Step 4: TP Optimization
Tuning Parameter Value

Fig. 2. Process of HwPC reduction, code region identification, and Tuning Parameter optimization.

F1-Score penalizes such behavior, encouraging both correctness and coverage.

ROC AUC measures discriminative ability across thresholds, it is crucial when suboptimal choices incur varying penalties. E.g., choosing 8 instead of 16 threads may cause only a 10% slowdown, while choosing 2 may cause an 80% slowdown. ROC AUC reflects the model's ability to rank configurations appropriately, not just classify them as optimal or not.

3. Application performance tuning using machine learning

This section presents a novel automatic methodology for tuning parallel applications using HwPCs, transforming raw HwPC data into actionable performance insights through ML methodologies. The methodology addresses performance analysis challenges by identifying code regions and optimizing TPs using HwPC data. The ensemble identifies both minimal sets of HwPCs required for code region identification and for TP optimization, improving efficiency and accuracy while automating application performance optimization.

In this work, a *code region* refers to a distinct computational block within an application that exhibits consistent performance characteristics and can be independently optimized. In an OpenMP context, code regions typically correspond to parallel loops or parallel sections enclosed by `#pragma omp parallel` directives. For GPU applications, code regions correspond to individual CUDA kernels. Each code region is characterized by its computational pattern (e.g., memory-bound, compute-bound, irregular access), which determines its optimal TP configurations. The methodology identifies code regions through their unique HwPC signatures rather than static code analysis.

A key motivation for using separate HwPC subsets for code region identification and TP optimization is that the most discriminative HwPCs for recognizing computational patterns could not be the most informative for predicting TPs. For example, cache-related HwPCs can distinguish compute- from memory-bound code regions, while instruction throughput HwPCs can be related to core utilization and thread placement TPs. By decoupling the subsets, the methodology maximizes predictive accuracy for both tasks.

In practice, as it will be discussed in Section 4, the computational overhead of the proposed approach remains modest: the reduction phase is performed offline, while online execution requires monitoring only 4-7 HwPCs per task, adding negligible runtime overhead.

Fig. 2 shows that the proposed methodology is composed of a **HwPC Reduction phase**, which is performed off-line only once for a specific architecture using the input dataset, and an **Application Execution phase**, which is applied to optimize a given application.

The input of the **HwPC Reduction Phase** is a dataset $\mathcal{D} = (\mathbf{l}, \mathbf{h}, \mathbf{t})$ where $\mathbf{l} \in \mathcal{L}$ represents code region labels, $\mathbf{h} \in \mathcal{H}$ denotes HwPC values, and $\mathbf{t} \in \mathcal{T}$ represents TP values. Its objectives are to minimize the sets of HwPCs needed to identify code regions, and optimizing TPs. Section 3.1 describes in detail this phase's two steps.

- **Step 1: Code Region HwPCs Reduction.** The $\mathbf{l}$ and $\mathbf{h}$ components of the dataset are used to identify relevant HwPCs for code region classification, producing a reduced HwPC list and a classification model.
- **Step 2: Tuning Parameter HwPCs Reduction.** The $\mathbf{l}$, $\mathbf{h}$, and $\mathbf{t}$ components of the dataset are used to identify relevant HwPCs for each TP associated to each code region, resulting in a reduced HwPC list and a prediction model for each specific TP of each code region.

The **Application Execution Phase** involves the optimization of a given application using the reduced sets of HwPCs obtained in the previous phase. It also includes two steps described in detail in Section 3.2.

- **Step 3: Code Region Identification.** At runtime, the values of the HwPCs from the list produced in Step 1 are collected, and the classification model is used to identify the executing code region.
- **Step 4: Tuning Parameter Optimization.** After identifying the executing code region, for each TP associated with this code region, it collects the values of the HwPCs of the list produced in Step 2, and the corresponding prediction model is used to determine the optimal TP value.

Consequently, the methodology's output is a value $t_{opt} \in \mathcal{T}$ for each TP that optimizes the performance of each code region in an application.

3.1. HwPC reduction phase

This subsection presents a dual-purpose methodology for HwPC reduction, as illustrated in Fig. 3. The first branch (Step 1) identifies important HwPCs for code region identification using sub-datasets containing code region labels and HwPC values. The second branch (Step 2) identifies important HwPCs for optimizing TPs using separate sub-datasets for each code region and TP combination, with their corresponding HwPCs.

The methodology uses an ML ensemble in three steps: (1) training and validation of classifier models to establish relationships between code regions and HwPCs (Step 1.1), and between TPs and HwPCs (Step 2.1); (2) extracting and quantifying importance scores for individual HwPCs across all ensemble models for code region identification (Step 1.2) and TP optimization (Step 2.2); and (3) ranking and reduction to identify a minimal HwPC set for code region identification (Step 1.3) and minimal specialized sets for TP optimization (Step 2.3). During inference, the models generate individual predictions with the final prediction determined by majority vote. Model performance is evaluated using accuracy to quantify correct classifications.

3.1.1. Ensemble training and validation

The ensemble training and validation correspond to Steps 1.1 and 2.1 in Fig. 3. The ensemble methodology integrates the four models in-

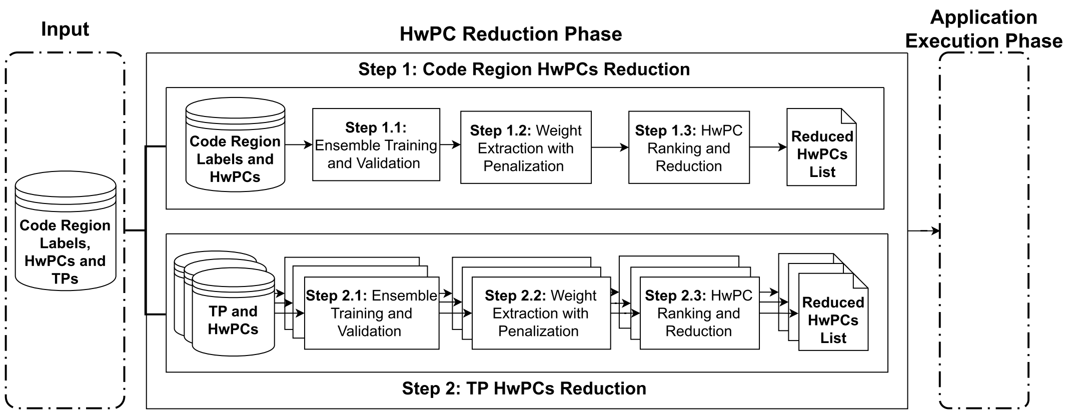
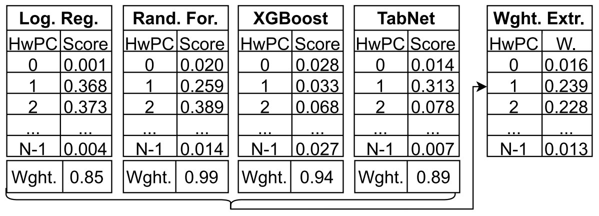

Input
HwPC Reduction Phase
Application Execution Phase
Step 1: Code Region HwPCs Reduction
Code Region Labels and HwPCs
Step 1.1: Ensemble Training and Validation
Step 1.2: Weight Extraction with Penalization
Step 1.3: HwPC Ranking and Reduction
Reduced HwPCs List
Code Region Labels, HwPCs and TPs
TP and HwPCs
Step 2.1: Ensemble Training and Validation
Step 2.2: Weight Extraction with Penalization
Step 2.3: HwPC Ranking and Reduction
Reduced HwPCs List
Step 2: TP HwPCs Reduction

Fig. 3. HwPCs reduction with model ensembles.



Log. Reg.		Rand. For.		XGBoost		TabNet		Wght. Extr.	
HwPC	Score	HwPC	Score	HwPC	Score	HwPC	Score	HwPC	W.
0	0.001	0	0.020	0	0.028	0	0.014	0	0.016
1	0.368	1	0.259	1	0.033	1	0.313	1	0.239
2	0.373	2	0.389	2	0.068	2	0.078	2	0.228
...	...	...	...	...	...	...	...	...	...
N-1	0.004	N-1	0.014	N-1	0.027	N-1	0.007	N-1	0.013
Wght.	0.85	Wght.	0.99	Wght.	0.94	Wght.	0.89		

Fig. 4. Step 2: Ensemble for HwPC importance scoring.

troduced in Section 2.4: Logistic Regression with Elastic Net, Random Forest, XGBoost, and TabNet, capturing linear and nonlinear relationships across varying dataset characteristics.

The implementation utilized a 70-30 split, with 30% reserved for final testing to provide an unbiased evaluation of generalization capabilities. The remaining 70% underwent 5-fold stratified cross-validation, preserving code region class distributions across folds. In each iteration, the models are trained on approximately 56% of the total data and validated on 14%. A value of $k = 5$ was chosen to balance statistical reliability and computational cost (only 5 training iterations). Larger k values (e.g., 10) doubled computational cost with marginal accuracy gains, whereas smaller values (e.g., 3) led to higher variance in validation scores. The stratified approach further ensured consistent handling of imbalanced TP distributions. Accuracy served as the primary evaluation metric throughout.

3.1.2. *Weight extraction*

The weight extraction process corresponds to Steps 1.2 and 2.2 in Fig. 3. Following training, each HwPC received normalized importance scores (0.0 to 1.0) based on model-specific criteria. Logistic Model with Elastic Net derives importance from coefficient magnitudes. Random Forest quantifies it through an average impurity reduction. XGBoost evaluates importance based on performance gain. TabNet employs an attention mechanism for feature contributions. The final importance score is computed using the weighted average of the individual model scores. Each model's weight corresponds to its validation accuracy.

Fig. 4 illustrates this process. The final weighted score W_i for the ith HwPC is calculated as:

$$W_i = \frac{\sum_{j=0}^{3} c_{i,j} w_j}{\sum_{j=0}^{3} w_j}, \tag{2}$$

where $c_{i,j}$ is a score of the ith HwPC from the jth model and w_j is an accuracy-based weight of the jth model. This formulation ensures that models with higher validation accuracy contribute more significantly to the final importance scores.

3.1.3. *HwPC ranking and reduction*

The HwPC ranking and reduction process corresponds to Steps 1.3 and 2.3 in Fig. 3. The kneedle algorithm [18] was used to select HwPCs for code region identification and TP optimization. The reduction process ranks HwPCs in descending order based on weighted average scores (Fig. 5(a)).

Then, as shown in Fig. 5(b), a reference line is established between points (0,0) and (N-1, accuracy N-1), where N is the total number of HwPCs and accuracy N-1 is the validation accuracy using all HwPCs. The models are trained iteratively, starting with the highest-ranked HwPC and progressively adding HwPCs in rank order. For each model, the distance between the accuracy and the reference line is calculated. The process stops when adding a new HwPC no longer increases this distance, indicating the *knee point* where accuracy gains become marginal. The minimum effective set of HwPCs includes those added up to this point, balancing model complexity with accuracy.

The reduction process produces a minimal list of HwPCs for code region identification, and TP-specific lists per code region for TP optimization:

- $HwPC_i = h_0, h_1, \ldots, h_{C-1}$: the minimal set of HwPCs to classify the code region i
- $HwPC_{ij} = h_0, h_1, \ldots, h_{T-1}$: the minimal set of HwPCs to tune the jth TP for the ith code region

where $0 < C, T \leq N - 1$.

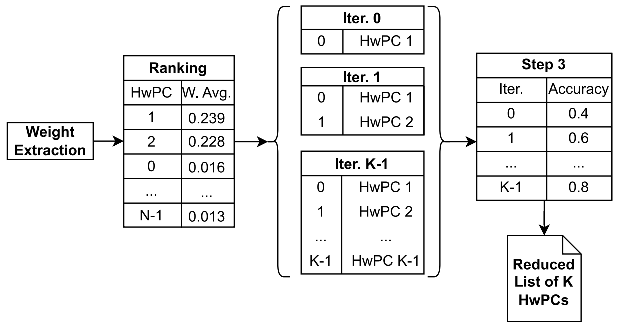
At the end of the reduction process, the final ensemble models trained with these reduced HwPC sets are stored. These ensembles are then used during the subsequent Application Execution Phase for code region identification and TP optimization.

3.2. *Application execution phase*

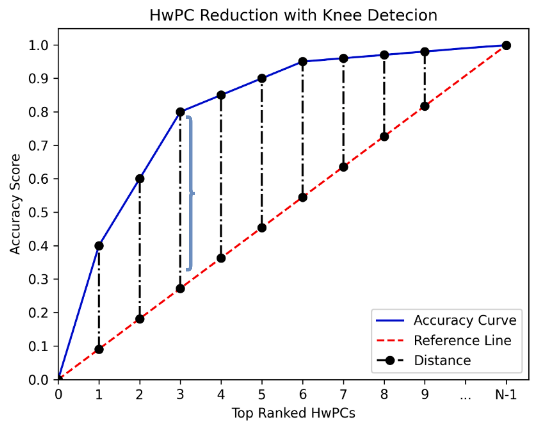
Fig. 6 delineates the process for code region identification and TP optimization during application execution. The process begins when the target application is running. First, Step 3 identifies different code regions within the application. In Step 3.1, the methodology collects measurements from a reduced set of HwPCs that were specifically selected to identify code regions.

In Step 3.2, this set of HwPCs' values is the input to the Code Region Identifier Model generated in the Step 1 of the HwPC reduction phase described in Section 3.1. The final code region class indicated by the ensemble is decided by majority vote, that is, whichever code region class receives the most votes.

As observed in Fig. 6, upon code region identification, the process transitions to Step 4, TP optimization. In Step 4.1, the system collects a specific set of HwPC measurements from the executing application. Importantly, each identified code region uses a different set of HwPCs



(a) Weight ranking and reduction process



(b) Knee detection of the accuracy curve

Fig. 5. Step 3: HwPC reduction considering the knee point.

for each TP being optimized. This means that for a given code region, the system monitors one specific set of HwPCs when optimizing TP A, and a different set of HwPCs when optimizing TP B. This approach allows for detailed analysis of how each individual TP affects performance within that specific code region.

In Step 4.2, the collected HwPC measurements are passed to the appropriate optimization models. For each code region, there are multiple TP optimization models - one for each TP. Each of these models was trained using the reduced set of HwPCs that are most relevant for optimizing that specific TP within that particular code region. As for the code region identification, each optimization model is an ensemble of the four ML classifiers and the final optimized TP value is determined by majority vote.

4. Evaluation

This section analyzes the results obtained using the proposed ensemble methodology for HwPC reduction, code region identification, and TP optimization. The evaluation methodology begins with a comprehensive analysis using OpenMP code regions on CPUs, where the reliability of the approach is validated through a comparative analysis with the findings in [1].

Sections 4.1 and 4.2 present this initial evaluation phase, where the dataset consisted of code regions already known to the models, providing a baseline for performance assessment. Section 4.3 demonstrates the robustness and generalization capabilities of the methodology by extending the evaluation to previously unseen code regions extracted from the NAS parallel benchmarks (NPB) [19]. Section 4.4 expands the analysis to GPU architectures, where the methodology was applied to GPU datasets containing different code regions.

Table 2
Preset HwPCs available on the Xeon E5-4620 platform.

Branches	Cache L1	Cache L2	Cache L3	TLB	Cycles	Ops.	Ins.
6	5	14	9	2	3	3	8

Table 3
Reduced HwPCs for code region identification.

Instructions	Operations	Branches
`PAPI_SR_INS`	`PAPI_FP_OPS`	`PAPI_BR_NTK`
`PAPI_FDV_INS`	`PAPI_DP_OPS`	`PAPI_BR_MSP`

Table 4
Reduced HwPCs for `2D4PStencil_E` code region's number of threads optimization.

L2 Cache	L3 Cache	Operations
`PAPI_L2_ICH`	`PAPI_L3_TCM`	`PAPI_FP_OPS`
`PAPI_L2_DCH`		
`PAPI_L2_DCR`		

All CPU benchmarks were compiled using GCC version 9.2.0 with -O2 optimization, and executed on the 32-core Xeon E5-4620 platform. GPU benchmarks were compiled with NVCC using CUDA 10.1 (driver 418.67) for GTX 1070 and RTX 2080, CUDA 12.1 (driver 535.183) for RTX 3090, and CUDA 12.3 (driver 545.23) for RTX 4080.

This cross-platform evaluation demonstrates the versatility of the methodology and its consistent effectiveness across both CPU and GPU architectures, showing its portability to diverse computing environments.

4.1. *Reduction of CPU HwPCs for identification and tuning of OpenMP regions*

The methodology was tested on an 18 code region dataset [3], comprising:

- **STREAM**: four code regions- `Copy`, `Scale`, `Sum`, and `Triad`-ach with distinct memory access patterns and operation counts [4].
- **PolyBench**: twelve code regions from synthetic benchmarks for common computational programs in scientific and engineering applications [5].
- **Additional Code Regions**: two code regions for computing Collatz sequences and Friendly numbers with different computational load per iteration.

PAPI was used for HwPC collection. It provides a standardized set of preset events for performance monitoring, which were grouped to maximize the information gathered during each execution while respecting hardware limitations [1]. Each HwPC group was measured across multiple executions, systematically varying parameters, including number of threads, thread affinity policy, scheduling policy, and chunk size. Problem sizes were computed using the methodology in [2] to stress different memory hierarchy levels. For statistical significance, 100 executions were conducted for each combination of HwPC group, problem size, and configuration. The preset HwPCs for the target architecture are shown in Table 2.

Starting with 50 available HwPCs, the methodology reduced the set to six HwPCs, shown in Table 3, for code region identification while maintaining high prediction capabilities.

The methodology further refined the reduction process at the TP level, creating unique HwPC sets for each code region's TPs to reflect distinct computational characteristics . Starting with 50 available HwPCs, the methodology successfully reduced the set to only 4–7 HwPCs for each TP. As an example, Table 4 shows the 5 HwPCs selected for optimizing the number of threads for the `2D4PStencil_E` code region.

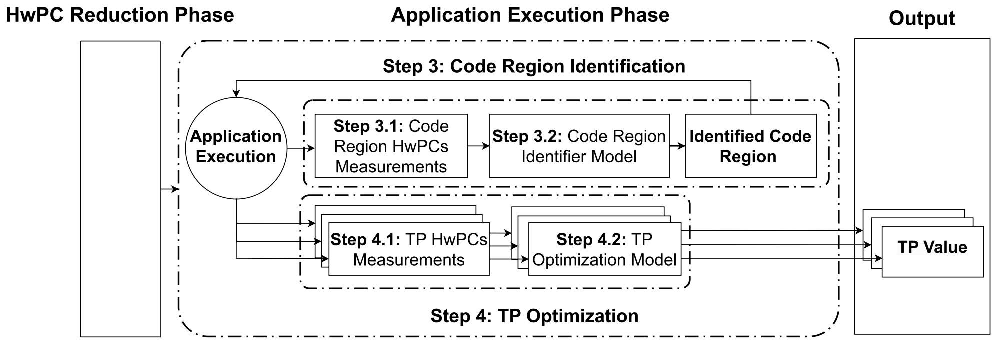

HwPC Reduction Phase
Application Execution Phase
Output
Step 3: Code Region Identification
Application Execution
Step 3.1: Code Region HwPCs Measurements
Step 3.2: Code Region Identifier Model
Identified Code Region
Step 4.1: TP HwPCs Measurements
Step 4.2: TP Optimization Model
TP Value
Step 4: TP Optimization

Fig. 6. Code region identification and tuning parameter optimization with model ensembles.

Table 5
Evaluation results of the ensemble for the code region identification (expanded benchmark.

Metric	K-F Acc.	Pr.	Rc.	F1	R-A	Full Set	Red.
Label	0.9765	0.9644	0.9556	0.9596	0.9996	50	6

4.2. Identification and tuning of OpenMP regions with a comprehensive dataset

Once we have a comprehensive dataset, the minimal set of HwPCs for code region identification, and the minimal sets of HwPCs for each TP, we can evaluate the effectiveness of the ensemble methodology. The methodology first goes through identifying the code regions with a minimal set of HwPCs. Once the code region is identified, it proceeds to optimize different TPs.

For comparison, the OpenTuner tool [20] was employed. It is an extensible auto-tuning framework that leverages multiple search techniques to efficiently explore configuration spaces. Its pluggable architecture enables sophisticated optimization strategies that support various hardware platforms by abstracting configuration spaces and search mechanisms.

4.2.1. Code region identification

This section evaluates the performance of the ensemble methodology for code region identification using minimal HwPC sets. The model was trained and validated using stratified 5-fold cross-validation on a subset of the data, achieving a mean accuracy of 97.65%. Its generalizability was then confirmed on a held-out test set comprising 30% of the full dataset.

The comprehensive results of this evaluation when applying the reduced HwPC set are presented in Table 5, which shows exceptional performance for all the metrics. The K-fold accuracy of 0.9765 and ROC AUC of 0.9996 indicate near perfect discrimination capability between different code regions. The precision (0.9644) and recall (0.9556) scores are well-balanced, yielding a strong F1-score of 0.9596 that confirms the methodology's robust predictive power. Notably, the ensemble achieved this high performance while dramatically reducing the required HwPC set from 50 to only 6 counters-an 88% reduction. This substantial dimensionality reduction, coupled with high classification accuracy, validates the effectiveness of the methodology and suggests strong generalizability for identifying relevant HwPCs for code region classification tasks.

4.2.2. Tuning parameter: Number of threads

This section evaluates the optimization of the number of threads TP for various code regions.

The evaluation starts considering the trade-off between performance (time) and resource (core) utilization. To quantify the number of threads efficiency, the performance index defined in [21] was used, which pro-

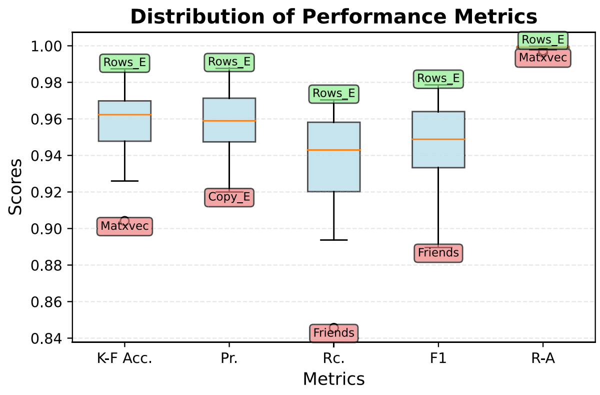


Fig. 7. Evaluation results of the ensemble for the prediction of the number of threads TP minimizing $P_i(X)$.

vides a metric for evaluating the number of threads efficiency. The performance index $P_i(X)$ was calculated using the following equation:

$$P_i(X) = \frac{T_t(X)}{E(X)} = \frac{X \cdot T_t(X)^2}{T_t(1)}, \tag{3}$$

where X denotes the number of threads, $T_t(X)$ and $E(X)$ denote the obtained execution time and efficiency of a code region using this number of threads. Therefore, $P_i(X)$ relates execution time with resource efficiency, allowing to automatically find the number of threads that maximizes performance (minimizing execution time) without wasting resources.

$P_i(X)$ captures the trade-off between execution time and thread utilization. Configurations with long execution times or poor efficiency are penalized, while configurations that achieve fast execution without wasting cores yield a lower index. Minimizing $P_i(X)$ identifies the optimal number of threads that balances runtime and resource usage. For example, if doubling the number of threads only slightly reduces execution time, efficiency decreases and $P_i(X)$ increases, signaling that adding more threads is not beneficial.

Fig. 7 presents the results of the ensemble methodology for optimizing the number of threads for all code regions, according to the objective function that minimizes $P_i(X)$. The results show that the ensemble achieves consistently high classification accuracy scores ranging from 0.9041 to 0.9974, while using only 4–7 HwPCs. The ensemble consistently reaches high precision and recall rates. The high ROC AUC scores (0.9963-0.9998) confirm robust discriminative capability across different code regions, demonstrating the ensemble's efficiency and practical applicability. These metrics indicate that the methodology reliably identifies the optimal number of threads' configurations for the code regions considered.

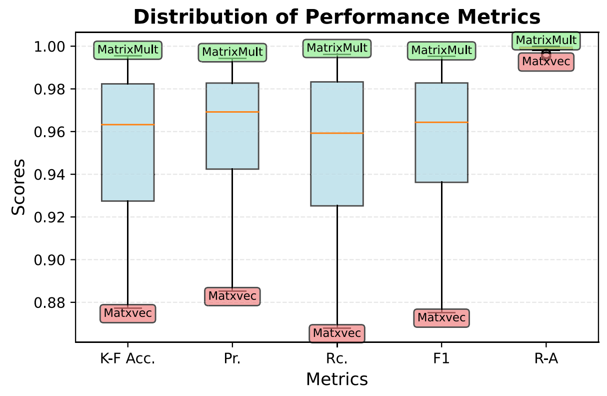


Fig. 8. Evaluation results of the ensemble for the prediction of the number of threads TP minimizing the execution time.

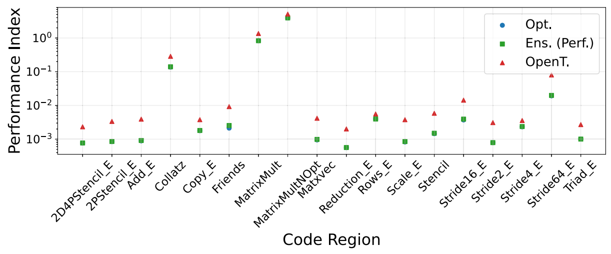


Fig. 9. Comparison of the optimal (Opt.), ensemble minimizing $P_i(X)$ (Perf), and OpenTuner (OpenT.) mean performance indices.

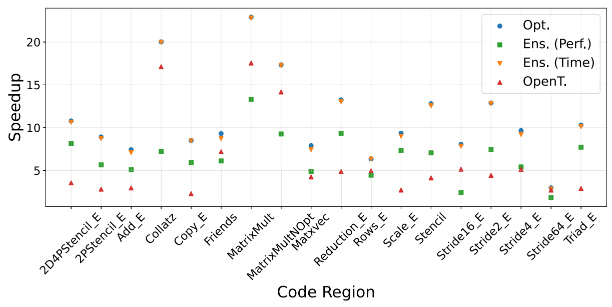


Fig. 10. Comparison of the optimal (Opt.), ensemble minimizing $P_i(X)$ (Perf), ensemble minimizing execution time (Time), and OpenTuner (OpenT.) mean speedups.

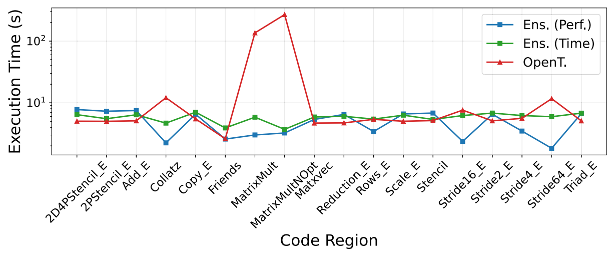


Fig. 11. Comparison of the execution times (logarithmic scale) required for computing the number of threads: ensemble minimizing $P_i(X)$ (Perf), ensemble minimizing execution time (Time), and OpenTuner (OpenT.).

For comparative analysis with OpenTuner, which optimizes execution time, a second ensemble model was trained using minimum execution time as the objective function. This alternative model not only provides a basis for a fair comparison, but also reinforces the adaptability of our proposed methodology to different optimization targets. Fig. 8 shows that the ensemble has consistently strong performance, maintaining high classification accuracy across all evaluation metrics, with K-fold accuracies ranging from 0.7987 to 0.9937. The high precision, recall, and F1-scores indicate that the approach effectively identifies optimal number of thread configurations for minimizing execution time with a reduced set of 4–7 HwPCs. This confirms the methodology's effectiveness to target execution time reduction.

Fig. 9 compares the optimal performance index (Eq. (3)) with the one obtained by the ensemble methodology (optimizing $P_i(X)$) and by OpenTuner (optimizing execution time). As expected, the ensemble methodology achieves optimal or almost optimal results for all regions, while OpenTuner consistently produces worse (i.e., higher) performance indices across all code regions, indicating less efficient resource utilization. Performance index degradation is particularly severe for certain code regions: `Add_E` (4.33× worse), `Scale_E` (4.22× worse), `Matxvec` (4.20× worse), and `Stride64_E` (4.05× worse).

Fig. 10 presents the speedup comparison between OpenTuner, the ensemble methodology (optimizing $P_i(X)$ and execution time), and the optimal speedup. It shows that the ensemble minimizing execution time (Time) achieves speedups closer to the optimal values in the majority of cases (13 out of 18 code regions), while the ensemble minimizing $P_i(X)$ (Perf) shows larger deviations from optimal values but it obtains better results than OpenTuner (11 out of 18).

Fig. 11 shows the comparison of the time needed to compute the number of threads by the ensemble and OpenTuner. The ensemble models consistently deliver 2-7 s when optimizing $P_i(X)$ and 3-7 s when optimizing speedup, ensuring low predictable optimization overhead in all scenarios. In contrast, OpenTuner exhibits significant time variability, with execution times ranging from 2 s to over 4 min.

This combination of generally superior speedup performance and consistent execution times reinforces the ensemble's practical advantage for performance-critical optimization scenarios.

4.2.3. Tuning parameter: Thread affinity policy

The next evaluated TP is thread affinity policy. The optimal affinity policy a^* for a given code region is defined as the policy that minimizes execution time:

$$a^* = \min_{a \in A} T(a) \tag{4}$$

where $A = \{\texttt{close}, \texttt{spread}\}$ is the set of available affinity policies and $T(a)$ represents the execution time under affinity policy a. The methodology predicts the affinity policy that achieves the minimum execution time. Prediction accuracy is measured as the percentage of correct identifications of a^*.

Fig. 12 shows that the ensemble methodology maintains high performance for affinity TP prediction. The ensemble achieves high K-fold accuracies ranging from 0.9512 to 0.9993, with most code regions exceeding 97% accuracy. The strong precision (0.9474-0.9991) and recall (0.9229-0.9990) values confirm reliable identification of optimal affinity configurations. The methodology accomplishes this performance using only 4–7 HwPCs. These results show the ensemble's effective feature reduction while maintaining strong discriminative capability for thread affinity optimization.

Fig. 13 shows that OpenTuner underperforms across all evaluation metrics for affinity tuning. K-fold accuracies range from 0.3427 to 0.6433, with precision, recall, and F1-scores consistently below 0.6, indicating poor classification performance barely exceeding random selection. Time percentage differences reveal substantial performance degradation, with most benchmarks showing deterioration up to 39.96%.

Finally, looking at the execution times between the ensemble and OpenTuner (Fig. 14), the ensemble shows a higher efficiency with consistently low execution times (0.42-0.59s) across all code regions. In contrast, OpenTuner exhibits highly variable performance, ranging from 0.79 s to under 1.5 min, with poor scalability for compute-intensive code regions like `MatrixMult` (32.91s) and `MatrixMultNOpt` (1m 33s). Over-

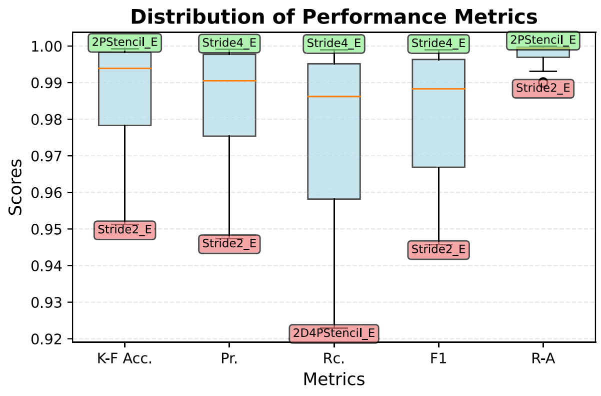


Fig. 12. Evaluation results of the ensemble for the prediction of the affinity TP.

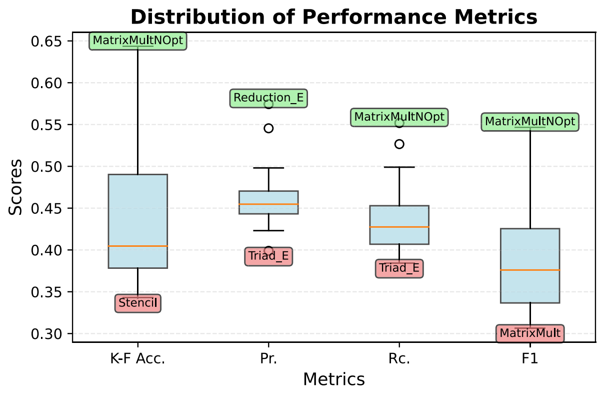


Fig. 13. Evaluation results of OpenTuner for the prediction of the affinity TP.

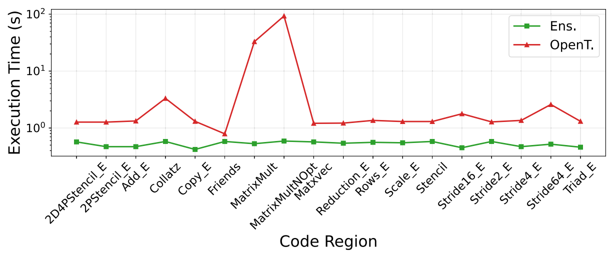


Fig. 14. Comparison of the execution times (logarithmic scale) required for computing affinity: ensemble and OpenTuner.

all, the ensemble methodology achieves from 2 to 180× faster optimization times while maintaining a higher classification performance reinforcing its practical advantage for runtime tuning of affinity TP.

4.2.4. Tuning parameter: Scheduling policy and chunk size

The final TPs we consider are the scheduling policy and the chunk size. The optimal scheduling policy s^* and corresponding chunk size c^* are determined by:

$$(s^*, c^*) = \min_{(s,c)\in S\times C} T(s,c) \tag{5}$$

where $S = \{\texttt{static}, \texttt{dynamic}, \texttt{guided}\}$ is the set of OpenMP scheduling policies and C is the discrete set of evaluated chunk sizes. The methodology predicts the policy-size combination that minimizes execution time. These TPs are evaluated through classification accuracy (correctly predicting the configuration with minimum execution time) since they represent discrete categorical choices rather than continuous resource allocation.

Table 6
Evaluation results of the ensemble for the prediction of the scheduling policy and chunk size parameter.

Tuning Param.	Reg./Metr.	K-F Acc.	Pr.	Rc.	F1	R-A	Red.
Scheduling Policy	`Collatz`	0.9883	0.9848	0.9831	0.9839	0.9992	5
	`Friends`	0.9893	0.9924	0.9917	0.9921	0.9994	6
Chunk Size	`Collatz`	0.9897	0.9913	0.9799	0.9853	0.9995	5
	`Friends`	0.9853	0.9824	0.9820	0.9821	0.9994	5

Table 7
Evaluation results of OpenTuner for the prediction of the scheduling policy and chunk size parameter.

Tuning Param.	Reg./Metr.	K-F Acc.	Pr.	Rc.	F1
Scheduling Policy	`Collatz`	0.4224	0.3702	0.3656	0.3654
	`Friends`	0.3637	0.3328	0.2813	0.2866
Chunk Size	`Collatz`	0.4165	0.3856	0.3921	0.3785
	`Friends`	0.3367	0.3556	0.3509	0.3341

Table 8
Comparison of the execution times required for computing scheduling policy and chunk size: ensemble and OpenTuner.

Reg./Exec. Time	Ens.	OpenT.
`Collatz`	1.34 s	4.18 s
`Friends`	1.61 s	0.78 s

Table 6 presents evaluation results for scheduling policy and chunk size TPs optimization on two code regions. The ensemble methodology demonstrates high performance for both TPs, achieving K-fold accuracies exceeding 98% across both cases. The results are consistently high across the rest of the presented metrics for both TPs. This shows that the ensemble can effectively identify optimal configurations for both TPs using only 4–7 HwPCs, demonstrating its versatility in handling diverse TPs beyond number of threads or thread affinity.

Table 7 summarizes the results that show that OpenTuner performs poorly for both scheduling policy and chunk size TP optimization. K-fold accuracies range from 0.3367 to 0.4224, indicating performance barely above random selection. Precision, recall, and F1-scores consistently remain below 0.4 across all cases, demonstrating inadequate classification capability for both TPs. Additionally, OpenTuner shows performance degradation with time percentage differences of 2.32-4.67%. Contrasting these results with the ensemble methodology's accuracy >98%, further confirms the ensemble's capabilities for TP optimization.

Finally, considering the execution times between the ensemble methodology and OpenTuner for scheduling and chunk size optimization (Table 8) the results show no clear time-wise advantage between them, as each method outperforms the other on different benchmarks. These divergent results highlight the complexity of scheduling and chunk size optimization.

4.3. Application to the NAS parallel benchmarks

To ensure a comprehensive and representative assessment, 7 OpenMP code regions from NAS Parallel Benchmarks (NPB) [19] were extracted. The code regions represent a diverse range of computational patterns. The NPB suite is widely recognized and extensively used in parallel computing to evaluate the performance of parallel systems. This dataset was not used for training, it was only used for evaluation purposes. As previously, PAPI was used to gather the HwPC data.

- `BT` **benchmark**: extracted code region - `add_BT`. It maps to the `BT`'s add function.
- `CG` **benchmark**: extracted code regions - `normztox_CG`, `norm_temps_CG`, `rhorr_CG`, `pr_beta_p_CG`, and `qAp_CG`.

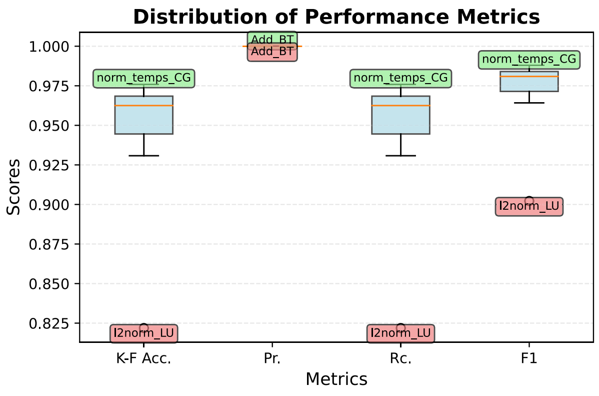


Fig. 15. Evaluation results of the ensemble for the code region identification (NAS benchmarks).

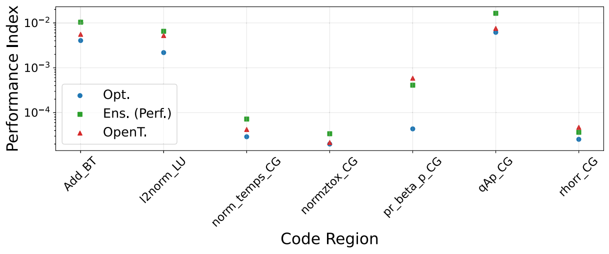


Fig. 16. Comparison of the optimal (Opt.), ensemble minimizing $P_i(X)$ (Perf), and OpenTuner (OpenT.) mean performance indices (NAS benchmarks).

- LU **benchmark**: extracted code region - l2norm_LU.

4.3.1. Code region identification

To validate the proposed methodology, we conducted comprehensive experiments using the NPB dataset. The initial evaluation focuses on code region identification by leveraging an ensemble of models trained on the comprehensive dataset from Section 4.2.

Fig. 15 presents code region identification results using the ensemble methodology with a minimal set of HwPCs. The ensemble demonstrates strong code region identification performance across most code regions, achieving high accuracy scores. The high F1 scores (>0.96) suggests the ensemble is highly confident of the predictions.

4.3.2. Tuning parameter: Number of threads

The evaluation continues with the optimization of the number of threads, focusing on efficiency as defined by the performance index (Eq. (3)).

Fig. 16 compares the optimal performance index (Eq. (3)) with the one obtained by the ensemble methodology (optimizing $P_i(X)$) and by OpenTuner (optimizing execution time). The results obtained by the ensemble are usually close to the optimal value, and, in all cases, similar to the ones obtained by OpenTuner. These results hint that, for these regions, the difference between the number of threads minimizing $P_i(X)$ and the one minimizing execution time is not significant.

Fig. 17 presents the speedup comparison between OpenTuner, the ensemble methodology (optimizing $P_i(X)$ and execution time), and the optimal speedup. It shows that, in effect, the execution time difference optimizing $P_i(X)$ and optimizing time are not significant in most cases (except for the qAp_CG case). Consequently, the values obtained by the ensemble methodology and OpenTuner are quite close.

Fig. 18 shows the comparison of the time needed to compute the number of threads by the ensemble and OpenTuner. The ensemble models consistently completes optimization in under a minute for all cases: 2-44 s when minimizing $P_i(X)$ and 2-50 s when maximizing speedup,

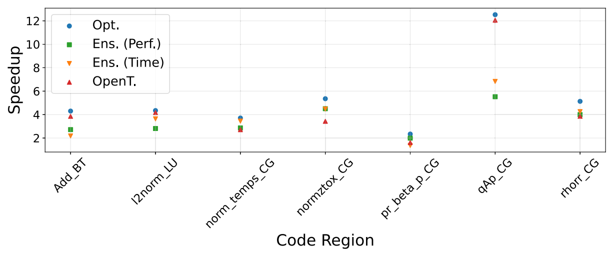


Fig. 17. Comparison of the optimal (Opt.), ensemble minimizing $P_i(X)$ (Perf), ensemble minimizing execution time (Time), and OpenTuner (OpenT.) mean speedups (NAS benchmarks).

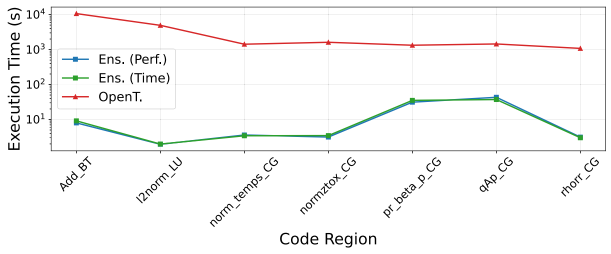


Fig. 18. Comparison of the execution times (logarithmic scale) required for computing the number of threads: ensemble minimizing $P_i(X)$ (Perf), ensemble minimizing execution time (Time), and OpenTuner (OpenT) (NAS benchmarks).

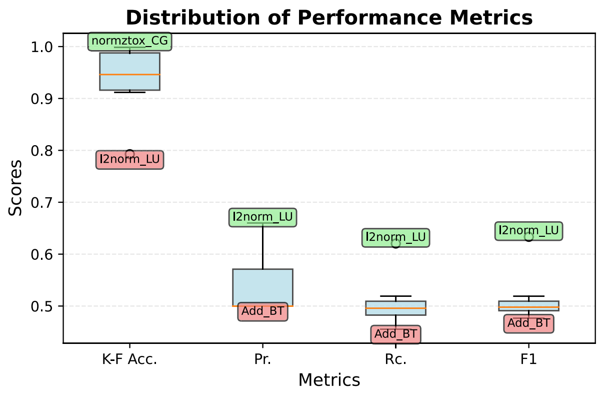


Fig. 19. Evaluation results of the ensemble for the prediction of the affinity TP (NAS benchmarks).

ensuring low predictable optimization overhead in all scenarios. In contrast, OpenTuner exhibits significant time variability, with execution times ranging from approximately 18 min for CG regions to nearly 3 h for BT regions. This means that the ensemble methodology is 20 to 300x faster than OpenTuner in these cases.

4.3.3. Tuning parameter: Thread affinity policy

Thread affinity policy is the next TP evaluated. After identifying the code region, a specialized set of HwPCs is used to predict the thread affinity policy. Fig. 19 shows varied performances across NAS benchmark code regions when using the ensemble methodology to predict optimal thread affinity. Several CG code regions exhibit high accuracy, e.g.: normztox_CG (0.9988) or rhorr_CG (0.9916), suggesting the ensemble methodology effectiveness for these sparse matrix operation patterns. However, precision, recall, and F1-scores for these code regions remained approximately 0.5, indicating potential imbalance in prediction classes.

Looking at the OpenTuner's affinity optimization results (Fig. 20) even more severe performance issues appear compared to the ensemble. The accuracy values are consistently poor across all code regions.

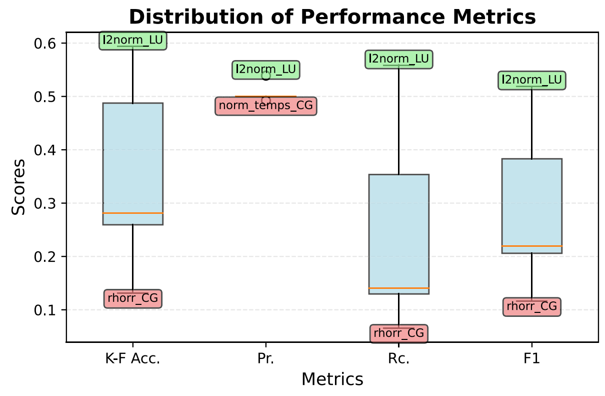


Fig. 20. Evaluation results of OpenTuner for the prediction of the affinity TP (NAS benchmarks).

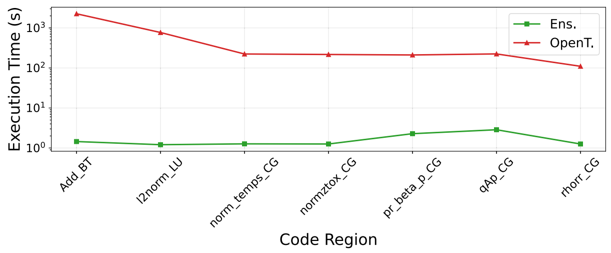


Fig. 21. Comparison of the execution times (logarithmic scale) required for computing affinity: ensemble and OpenTuner (NAS benchmarks).

As before, OpenTuner suffers from the same class imbalance symptoms with multiple code regions having low precision and recall values.

Finally, the execution times (Fig. 21) for optimal thread affinity prediction between the ensemble methodology and OpenTuner are compared. While the ensemble consistently completes optimization in 1-3 s across all code regions, OpenTuner requires dramatically longer times ranging from ∼2-4 min for CG code regions up to nearly 38 min for BT code regions. Overall, despite both approaches struggling with prediction accuracy due to class imbalance, the ensemble achieves comparable or better performance metrics while being orders of magnitude faster than OpenTuner's long optimization times.

4.4. Applying the methodology to GPUs

The final experiments demonstrate the methodology's applicability to GPUs by reducing necessary HwPCs for code region identification and TP optimization. HwPC data were collected using the Kernel Tuning Toolkit (KTT) [22,23] on GTX 1070 (Pascal), RTX 2080 (Turing), RTX 3090 (Ampere), and RTX 4080 (Ada Lovelace) GPUs across the code regions analyzed by Petrovič et al. [23]: `Convolution`, `Couluomb Sum`, `N-body`, `Transposition`, and `GEMM`. Additionally code regions were added to some of the architectures. The `Reduction` code region was added for Pascal, the `Biconjugate Gradient` and `Hotspot` code regions were added for Turing, and `Biconjugate Gradient` was added for Ada Lovelace. All GPU code regions use CUDA as the parallelization model.

A summary of the TPs targeted per architecture is presented in Table 9, where **work-group size** balances parallelism against resource consumption, **work-item coarsening** adjusts the computational workload per thread, **local memory caching** enables explicit use of fast shared memory as a cache, **private memory caching** optimizes register usage for fastest data access, **tile size** defines memory blocking dimensions to improve locality, **loop unrolling** reduces branching overhead and increases instruction-level parallelism, **local memory padding** prevents

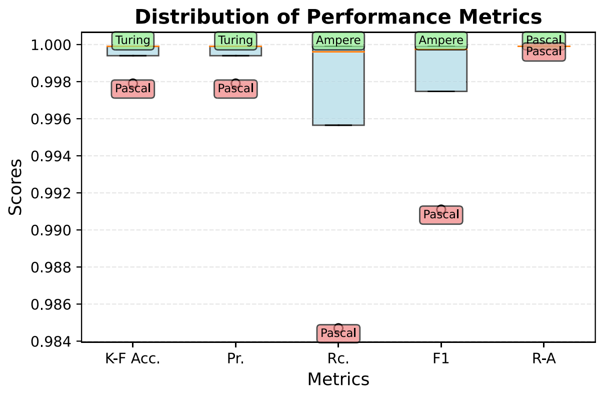


Fig. 22. Evaluation results of the ensemble for the HwPC reduction and code region identification for GPUs.

bank conflicts in GPU shared memory, and explicit **vectorization** facilitates generation of efficient vector memory instructions.

The ensemble methodology was able to greatly reduce the number of HwPCs for code region identification as exemplified in Table 10 for the GTX 1070.

As summarized in Fig. 22 stratified 5-fold cross-validation achieved average accuracy scores of 0.9999 for the four datasets, with near-perfect results across precision, recall, F1-Score, and ROC AUC metrics. This exceptional accuracy suggests highly effective prediction despite significantly different available HwPCs between architectures (32 vs. 167), although the small dataset size and limited number of code regions may contribute to these results.

Finally, Table 11 summarizes code region-specific TP optimization across the four GPU architectures. In this case we are presenting averaged prediction metrics for different code regions, since each contains varying numbers of TPs. For each code region we also present the range of HwPCs required for the optimization of the TPs. The results demonstrate consistently strong performance across all architectures, with most code regions achieving high accuracy (> 0.81), precision (> 0.81), and recall (>0.81) values.

Pascal generally exhibits the strongest performance, particularly evident in the `Reduction` code region which achieves near-perfect metrics (0.9933 accuracy, F1 score of 0.9937). Turing shows good performance with notable strength in `N-body` (0.9200 accuracy) and `BiCG` (0.9307 accuracy) code regions. Ampere demonstrates solid performance across most code regions, although with slightly lower accuracy in `Convolution` and `Coulomb Sum` compared to other architectures. Ada Lovelace shows improved performance in `Coulomb Sum` (0.8775 accuracy) compared to Ampere, while maintaining fair results for other code regions. Across all architectures, the `Coulomb Sum` kernel consistently presents the most challenging optimization case, due to the extremely small sample size (209–270 samples). Meanwhile, `N-body` and specialized code regions like `Reduction` and `BiCG` achieve the highest prediction accuracy.

Notably, the ensemble requires only a modest number of HwPCs to achieve these predictions, with most code regions needing between 2–9 HwPCs, indicating efficient feature utilization for accurate TP optimization across different GPU generations. These results validate that the ensemble methodology maintains robust optimization capabilities across diverse GPU architectures.

4.5. Discussion of the evaluation results

The evaluation results demonstrated both strengths and limitations of the ensemble methodology for optimizing OpenMP code region TPs. The methodology predicted near-optimal number of threads across diverse code regions, with K-fold accuracy scores frequently exceeding

Table 9
Common TPs by code region. Tile size is ticked when it can be configured differently than WG size. Abbreviations used: WG is work-group, LM is local memory, PM is private memory.

Reg./Metr.	WG size	coarsen.	LM cach.	PM cach.	Tile size	unroll.	LM pad.	vector.
`Conv.`	×	×	×	×	×	×	×	×
`Coul. Sum`	×	×				×		×
`N-body`	×	×	×	×		×		×
`Transp.`	×	×	×		×		×	×
`GEMM`	×	×	×		×	×		×
`Reduc.`	×	×						×
`BiCG`	×	×	×		×	×		
`Hotspot`	×	×	×		×	×		

Table 10
List of the reduced HwPCs for the Pascal GPU.

GTX 1080 (Pascal)
`inst_per_warp`
`gld_requested_throughput`
`gst_requested_throughput`
`dram_read_throughput`

Table 11
Results for code region TP evaluation on different GPUs.

GPU	Reg./Metr.	K-F Acc.	Pr.	Rc.	F1	R-A	Red.
Pascal	`Conv.`	0.9024	0.9243	0.9061	0.9115	0.9832	2–8
	`Coul. Sum`	0.7930	0.7876	0.7819	0.7824	0.8272	2–8
	`N-body`	0.9166	0.9153	0.8976	0.8999	0.9670	2–8
	`Transp.`	0.8690	0.8633	0.8607	0.8597	0.9734	6–11
	`GEMM`	0.9035	0.9062	0.8972	0.8997	0.9546	3–11
	`Reduc.`	0.9933	0.9933	0.9952	0.9937	0.9999	2
Turing	`Conv.`	0.8351	0.8694	0.8244	0.8383	0.9660	4–8
	`Coul. Sum`	0.7879	0.7820	0.7790	0.7783	0.8283	4–9
	`N-body`	0.9200	0.9224	0.9085	0.9129	0.9819	3–6
	`Transp.`	0.8411	0.8412	0.8393	0.8376	0.9588	5–8
	`GEMM`	0.8673	0.8652	0.8549	0.8585	0.9334	3–8
	`BiCG`	0.9307	0.9293	0.9273	0.9280	0.9286	3–9
	`Hotspot`	0.8859	0.8968	0.8860	0.8879	0.9740	4–8
Ampere	`Conv.`	0.8163	0.8433	0.8101	0.8189	0.9621	3–7
	`Coul. Sum`	0.7244	0.7151	0.7156	0.7119	0.8071	2–7
	`N-body`	0.9126	0.9167	0.9008	0.9057	0.9783	3–6
	`Transp.`	0.8425	0.8433	0.8409	0.8391	0.9558	5–7
	`GEMM`	0.8455	0.8458	0.8323	0.8377	0.9238	4–7
Lovelace	`Conv.`	0.8197	0.8421	0.8068	0.8142	0.9642	2–7
	`Coul. Sum`	0.8775	0.8718	0.8675	0.8676	0.9211	5–7
	`N-body`	0.9104	0.9104	0.9046	0.9069	0.9815	2–9
	`Transp.`	0.8166	0.8225	0.8102	0.8128	0.9495	4–7
	`GEMM`	0.8666	0.8669	0.8618	0.8633	0.9540	3–6
	`BiCG`	0.9235	0.9235	0.9235	0.9235	0.9254	2–9

0.90 and ROC AUC scores consistently exceeding 0.99. However, performance varied across code regions, with operations such as `Copy_E` and `Matxvec` showing lower accuracy scores (0.9260 and 0.9041, respectively), indicating potential areas for improvement.

Thread affinity prediction demonstrated strong performance, with most code regions achieving accuracy scores greater than 0.95 while requiring only 4–7 HwPCs, suggesting effective optimization with minimal overhead. Compared to OpenTuner, the ensemble methodology demonstrated a crucial advantage in optimization time. While OpenTuner required higher optimization time-notably 32.91 s for `MatrixMult` and 1 min 33 s for `MatrixMultNOpt`-the ensemble methodology consistently completed optimization in slightly over 50 s across all evaluated code regions.

The methodology demonstrated versatility in handling other OpenMP optimization aspects, with strong performance in predicting scheduling policies and chunk sizes for the `Collatz` and `Friends` code regions (accuracies above 0.90). However, limitations include varying performance across code regions and certain complex computational patterns, indicating that the current HwPC set may not fully capture all relevant program behavior aspects. Results highlight a trade-off between optimization quality and speed, not always achieving the performance of search methods like OpenTuner, the ensemble methodology delivers substantial speedups with minimal optimization overhead, making it suitable for dynamic optimization scenarios requiring rapid adaptation to changing code regions.

Application to NAS Parallel Benchmarks further validated the methodology's effectiveness across scientific code regions, maintaining strong performance in code region identification with most code regions showing accuracy above 0.90. Although the ensemble methodology generally achieved near-optimal speedups for number of threads and maintained reasonable accuracy in thread affinity prediction, its most significant advantage was computational efficiency. The ensemble consistently completed optimization tasks in seconds compared to OpenTuner's minutes to hours, representing execution time improvements of up to $\times 30$ while delivering comparable or superior optimization quality. The GPU validation experiments on GTX 1070 (Pascal), RTX 2080 (Turing), RTX 3090 (Ampere), and RTX 4080 (Ada Lovelace) architectures demonstrated the methodology's cross-platform effectiveness across four GPU generations. TP optimization across multiple code regions showed consistently strong performance (>0.81 accuracy, precision, and recall), with Pascal achieving the highest performance, particularly in specialized kernels like `Reduction` (0.9933 accuracy). Turing and Ada Lovelace demonstrated competitive results, while Ampere showed solid performance across most kernels with some variations in specific code regions. The methodology required only 2–9 HwPCs for most code regions, validating efficient feature utilization and broad applicability across modern GPU architectures in heterogeneous computing environments.

Finally, we analyze the computational cost of our methodology. The initial reduction phase, which involves automated HwPC selection and model training, incurs a high but one-time cost per target architecture. The application execution phase, which occurs frequently, introduces minimal overhead. This overhead consists of HwPC data collection and model inference, and has been measured on the same platform used in the experimentation (Xeon E5-4620), obtaining:

- Code Region Identification:
  - Collection: ∼3 it./region, $\bar{t} = 0.1063ms$/iter ($\sigma = 0.0279$ ms)
  - Inference: $\bar{t} = 0.5667ms$ ($\sigma = 0.0270ms$).
- Tuning Parameter (TP) Optimization:
  - Collection: ∼2-3 it./region, $\bar{t} = 0.0942ms$/iter ($\sigma = 0.0293ms$).
  - Inference: $\bar{t} = 0.8404ms$ ($\sigma = 0.0122ms$).

The results demonstrate that the runtime overhead of our methodology during application execution is low, making it suitable even for dynamic tuning.

5. Related work

ML methodologies for characterizing code regions and tuning parallel applications using HwPCs have gained significant attention recently, spanning various approaches for application identification and parameter optimization across CPU and GPU architectures.

The foundation for HwPC-based optimization lies in standardized access to performance monitoring capabilities. The PAPI project [7,8] established the foundational infrastructure by specifying a standard API for accessing HwPCs across diverse microprocessor architectures, providing cross-platform access to the small set of registers that count processor events. This standardization enables correlation between source code structure and architectural mapping efficiency, facilitating performance analysis and tuning across major HPC platforms. However, the reliability of HwPC measurements presents significant challenges for optimization methodologies. Weaver and McKee [24] demonstrated that HwPCs can exhibit coefficients of variation up to 1.07% under standard conditions, though careful experimental setup can reduce observed errors to less than 0.002%. Their analysis revealed that subtle changes in experimental conditions can significantly impact results, highlighting the importance of rigorous measurement protocols for HwPC-based optimization approaches.

For CPU-based parallel applications, several methodologies have emerged leveraging different aspects of performance data. The authors in [2] predicted optimal OpenMP number of threads using HwPCs and correlation analysis, addressing imbalanced datasets through Random Forest and binary classification. Alternatively, Yadav et al. [25] employed Random Forest Regression on static code features rather than HwPCs, analyzing loop characteristics to optimize thread numbers. OpenTuner [20] provides a broader optimization framework using ensembles of search techniques-including AUC Bandit Meta, differential evolution, and hillclimbers-to efficiently explore configuration spaces across computational domains. Dutta et al. [26] introduced an OpenMP loop auto-tuning approach using Graph Neural Networks with flow-aware program representation and HwPC data integration.

Beyond runtime optimization, HwPCs have proven valuable for compile-time improvements. Wicht et al. [27] developed a Profile-Guided Optimization approach that samples Last Branch Record HwPCs to recreate source locations, achieving 83% of the gains obtained with instrumentation-based PGO while reducing profiling overhead from 16% to only 1.06%.

For GPU-based applications, the optimization landscape presents unique challenges due to architectural diversity and varying data characteristics. Filipovič et al. [28] introduced a method that leverages HwPCs to navigate autotuning search spaces towards faster GPU implementations. Their approach builds problem-specific models from sampled tuning spaces that can be applied across various GPUs and input characteristics. Similarly, the authors also introduced Kernel Tuning Toolkit [23], which combines static analysis and ML to optimize parallel code regions across programming models and architectures. Conversely, the Kernel Tuner [29] offers a Python-based tool supporting various programming languages and search algorithms for both compile-time and runtime optimization.

While these studies demonstrate the potential of data-driven approaches for parallel computing optimization, current approaches face several limitations. The reliability concerns identified by Weaver and McKee [24] necessitate careful experimental design, while the infrastructure provided by PAPI enables standardized access but does not address the challenge of selecting the most informative HwPCs from the hundreds available on modern processors. Furthermore, existing GPU optimization approaches, such as Filipovič et al. [28] focus on autotuning convergence but do not provide comprehensive automated methodologies for identifying the most relevant HwPCs across diverse architectures.

In contrast to the works discussed, the core contribution of our research is a novel methodology to derive minimal sets of HwPCs for both code region identification and tuning parameter optimization. This methodology forms the foundation of our proposed end-to-end optimization process, but its principled approach to feature selection is also directly applicable as a complement to other HwPC-based techniques, enabling them to identify a more relevant and efficient set of counters.

6. Conclusions and future work

This paper presents an automated ML methodology for performance optimization in heterogeneous HPC environments that addresses critical challenges in utilizing HwPCs for application tuning. The proposed methodology integrates HwPC reduction, parallel code region characterization, and TP optimization while demonstrating efficiency and portability across architectures. The methodology advances automated performance optimization by: (1) addressing HwPC quantity versus accessibility through ML-based identification of minimal HwPC sets, reducing data collection overhead while maintaining precision; (2) tackling data interpretation using ML ensemble methodologies that automatically process complex HwPC data patterns; and (3) enabling efficient TP optimization through fast automated analysis. Experimental validation across diverse hardware architectures and code regions demonstrates high accuracy in predicting optimal TP configurations, and architecture-agnostic design with consistent performance across CPU and GPU platforms.

Future research directions include: (1) Integration of the proposed methodology into a dynamic tuning environment that monitors execution of parallel applications, identifies code regions and adjusts their TPs continuously during runtime; (2) Verification of the proposed methodology for dynamically tuning a wider set of parallel applications during runtime; (3) Define multi-objective optimization methodology that simultaneously tunes multiple TPs while balancing conflicting performance objectives such as execution time, energy consumption, and resource utilization; (4) Evaluation of the methodology on non-NVIDIA GPU platforms (e.g., AMD GPUs) to assess the generalizability of the approach across diverse hardware vendors and generations; (5) Apply the proposed methodology for different purposes for GPU platforms, such as developing visualization tools that help developers understand how different TPs affect HwPC behavior and overall application performance; and (6) Scalability studies for large-scale systems. These directions aim to expand the methodology's capabilities while maintaining automation, portability, and efficiency as HPC systems evolve in complexity and heterogeneity.

CRediT authorship contribution statement

Suren Harutyunyan Gevorgyan: Writing – review & editing, Writing – original draft, Investigation; **Eduardo César:** Writing – review & editing, Writing – original draft, Investigation; **Anna Sikora:** Writing – review & editing, Writing – original draft, Investigation; **Jiří Filipovič:** Writing – review & editing, Data curation; **Jordi Alcaraz:** Writing – review & editing, Data curation.

Data availability

Data will be made available on request.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Funding

This work was supported by the Ministerio de Ciencia e Innovación MCIN AEI/10.13039/501100011033 under contract PID2023-146193OB-I00, by the Catalan government under contract 2021 SGR 00574, and by the Ministry of Education, Youth and Sports from the Large Infrastructures for Research, Experimental Development and Innovations project “e-Infrastructure CZ - LM2023054”.

References

[1] J. Alcaraz, A. Sikora & E. César, Hardware Counters' Space Reduction for Code Region Characterization, Euro-Par 2019: Parallel Processing, 2019, Springer International Publishing, Cham, 74–86.

[2] J. Alcaraz, A. TehraniJamsaz, A. Dutta, A. Sikora, A. Jannesari, J. Sorribes & E. César, Predicting number of threads using balanced datasets for OpenMP regions, Computing, 105 (5) (2023) 999–1017. https://doi.org/10.1007/S00607-022-01081-6

[3] S. Harutyunyan, E. César, A. Sikora, J. Filipović, A. Dutta, A. Jannesari & J. Alcaraz, Efficient Code Region Characterization Through Automatic Performance Counters Reduction Using Machine Learning Techniques. Carretero, J.,Shende, S., Garcia-Blas, J.,Brandic, I., Olcoz, K.,Schreiber, M. (EDS.), Euro-Par 2024: Parallel Processing, 2024, Springer Nature Switzerland, Cham, 18–32.

[4] J. McCalpin, Memory Bandwidth and Machine Balance in High Performance Computers, IEEE Technical Committee on Computer Architecture Newsletter (1995) 19–25.

[5] T. Yuki, Understanding PolyBench/C 3.2 kernels, in: S. Rajopadhye, S. Verdoolaege (Eds.), Proceedings of the 4th International Workshop on Polyhedral Compilation Techniques, the 4th International Workshop on Polyhedral Compilation TechniquesVienna, Austria, 2014.

[6] J. Alcaraz, S. Sleder, A. Tehrani Jamsaz, A. Sikora, A. Jannesari, J. Sorribes & E. César, Building Representative and Balanced Datasets of OpenMP Parallel Regions, 29th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP 2021), Valladolid, Spain, March 10–12, 2021, IEEE (2021) 67–74. https://doi.org/10.1109/PDP52278.2021.00019

[7] H. Jagode, A. Danalis, G. Congiu, D. Barry, A. Castaldo, J. Dongarra, Advancements of PAPI for the exascale generation, Int. J. High Perform. Comput. Appl. 39 (2) (2025) 251–268. https://doi.org/10.1177/10943420241303884

[8] S. Browne, J. Dongarra, N. Garner, K. London, P. Mucci, A scalable cross-platform infrastructure for application performance tuning using hardware counters, in: SC '00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing, 2000, p. 42. https://doi.org/10.1109/SC.2000.10029

[9] D. Opitz, R. Maclin, Popular ensemble methods: an empirical study, J. Artif. Int. Res. 11 (1) (1999) 169–198.

[10] R. Polikar, Ensemble based systems in decision making, IEEE Circuits Syst. Mag. 6 (3) (2006) 21–45.

[11] D.R. Cox, The regression analysis of binary sequences, J. R. Stat. Soc. Ser. B 20 (2) (1958) 215–232.

[12] H. Zou, T. Hastie, Regularization and variable selection via the elastic net, J. R. Stat. Soc. Ser. B 67 (2) (2005) 301–320.

[13] R. Tibshirani, Regression shrinkage and selection via the lasso, J. R. Stat. Soc. Ser. B 58 (1) (1996) 267–288.

[14] A.E. Hoerl, R.W. Kennard, Ridge regression: biased estimation for nonorthogonal problems, Technometrics 12 (1) (1970) 55–67.

[15] L. Breiman, Random forests, Mach. Learn. 45 (2001) 5–32.

[16] T. Chen, C. Guestrin, XGBoost: a scalable tree boosting system, in: ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '16, New York, NY, USA, ACM, 2016, pp. 785–794. Proceedings of the 22nd.

[17] S.O. Arik, T. Pfister, TabNet: Attentive Interpretable Tabular Learning, CoRR abs/1908.07442, 2019.

[18] V. Satopaa, J. Albrecht, D. Irwin, B. Raghavan, Finding a "Kneedle" in a Haystack: detecting knee points in system behavior, in: 2011 31st International Conference on Distributed Computing Systems Workshops, 2011.

[19] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, S. Weeratunga, The NAS parallel benchmarks, Int. J. Supercomput. Appl. 5 (3) (1991) 63–73. https://doi.org/10.1177/109434209100500306

[20] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'reilly, S. Amarasinghe, OpenTuner: an extensible framework for program autotuning, in: Proceedings of the 23rd International Conference on Parallel Architectures and Compilation, PACT '14, the 23rd International Conference on Parallel Architectures and Compilation, PACT '14New York, NY, USA, Association for Computing Machinery, 2014, pp. 303–316. https://doi.org/10.1145/2628071.2628092

[21] E. Cesar, A. Moreno, J. Sorribes, E. Luque, Modeling master/worker applications for automatic performance tuning, Parallel Comput. 32 (7) (2006) 568–589. Algorithmic Skeletons. https://doi.org/10.1016/j.parco.2006.06.005

[22] F. Petrovič, J. Filipovič, Kernel Tuning Toolkit 22 (2023) 101385.

[23] F. Petrovič, D. Střelák, J. Hozzová, J. Oľha, R. Trembecký, S. Benkner, J. Filipovič, A benchmark set of highly-efficient CUDA and OpenCL kernels and its dynamic autotuning with kernel tuning toolkit, Future Gener. Comput. Syst. 108 (2020) 161–177.

[24] V.M. Weaver, S.A. Mckee, Can hardware performance counters be trusted?, in: IEEE International Symposium on Workload Characterization, 2008, pp. 141–150. https://doi.org/10.1109/IISWC.2008.4636099

[25] A. Yadav, M. Ahmed, Predictive modeling for thread optimization in OpenMP-based parallelization using machine learning, in: 2024 IEEE 17th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC), 2024, pp. 339–344. https://doi.org/10.1109/MCSoC64144.2024.00062

[26] A. Dutta, J. Alcaraz, A. Tehrani, A. Jamsaz, E. Sikora, A. Cesar, Jannesari, Pattern-based autotuning of OpenMP loops using graph neural networks, in: 2022 IEEE/ACM International Workshop on Artificial Intelligence and Machine Learning for Scientific Applications (AI4S), 2022, pp. 26–31.

[27] B. Wicht, R.A. Vitillo, D.D. Chen, Hardware counted profile-guided optimization, arXiv:1411.6361. (2014).

[28] J. Filipovič, J. Hozzová, A. Nezarat, J. Ol'ha, F. Petrovič, Using hardware performance counters to speed up autotuning convergence on GPUs, J. Parallel Distrib. Comput. 160 (2022) 16–35. https://doi.org/10.1016/j.jpdc.2021.10.003

[29] B.V. Werkhoven, Kernel tuner: a search-optimizing GPU code auto-tuner, Future Gener. Comput. Syst. 90 (2019) 347–358. https://doi.org/10.1016/j.future.2018.08.004