Language-Based Deployment Optimization for Random Forests (Invited Paper)

Jannik Malcher
TU Dortmund
Germany
jannik.malcher@tu-dortmund.de

Daniel Biebert
TU Dortmund
Germany
daniel.biebert@tu-dortmund.de

Kuan-Hsun Chen
University of Twente
Netherlands
k.h.chen@utwente.nl

Sebastian Buschjäger
Lamarr Institute for Machine Learning and Artificial Intelligence
Germany
sebastian.buschjaeger@tu-dortmund.de

Christian Hakert
TU Dortmund
Germany
christian.hakert@tu-dortmund.de

Jian-Jia Chen
TU Dortmund
Germany
Lamarr Institute for Machine Learning and Artificial Intelligence
Germany
jian-jia.chen@tu-dortmund.de

Abstract

Arising popularity for resource-efficient machine learning models makes random forests and decision trees famous models in recent years. Naturally, these models are tuned, optimized, and transformed to feature maximally low-resource consumption. A subset of these strategies targets the model structure and model logic and therefore induces a trade-off between resource-efficiency and prediction performance. An orthogonal set of approaches targets hardware-specific optimizations, which can improve performance without changing the behavior of the model. Since such hardware-specific optimizations are usually hardware-dependent and inflexible in their realizations, this paper envisions a more general application of such optimization strategies at the level of programming languages. We therefore discuss a set of suitable optimization strategies first in general and envision their application in LLVM IR, i.e. a flexible and hardware-independent ecosystem.

CCS Concepts: • **Software and its engineering → Compilers**; **Abstraction, modeling and modularity**; *Embedded software*; • **Computing methodologies → Ensemble methods**.

Keywords: Optimization, Random Forests, LLVM IR

LCTES '24, June 24, 2024, Copenhagen, Denmark

ACM ISBN 979-8-4007-0616-5/24/06
https://doi.org/10.1145/3652032.3659366

ACM Reference Format:
Jannik Malcher, Daniel Biebert, Kuan-Hsun Chen, Sebastian Buschjäger, Christian Hakert, and Jian-Jia Chen. 2024. Language-Based Deployment Optimization for Random Forests (Invited Paper). In *Proceedings of the 25th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '24), June 24, 2024, Copenhagen, Denmark.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3652032.3659366

1 Introduction

Efforts towards the performance optimization of random forest ensembles span a broad field, from modified training algorithms (see. e.g., [4] and references therein), over to implementation-based approaches (see e.g. [2, 5, 7–9]) up to the design of hardware acceleration (see, e.g., [3, 10]).

Implementation-based optimizations are particularly attractive as they do not change a given forest and preserve its predictive performance (compared to, e.g., modified training algorithms) while they can readily be exploited on existing hardware without the need to build new hardware (compared to, e.g., special hardware accelerators). To do so, modifications of the implementation, i.e., in the transformation from a logical model to high-level source code, to intermediate representations, or to machine code are applied, such that an improvement in the overall model performance is gained. While such modifications can be hooked at various levels, they usually work on a low level to gain explicit hardware control. A widely unexplored direction is how to apply modifications independent of the concrete underlying machine code on a higher language level.

The approach envisioned in this position paper is to employ model-agnostic implementation optimizations to random forest models within various layers of the compiler stack. We propose the LLVM as a premier and machine-independent target. In this paper, we present several optimization strategies for random forest implementations, first

in an abstract manner in Section 2. Afterwards, we envision their unified integration into the LLVM ecosystem in Section 3 We further envision how to set the optimization strategies into a broader scope of the in-the-loop optimization within the LLVM ecosystem.

2 Random Forest Optimization Strategies

As a first and important step towards the envisioned model agnostic implementation optimization, different optimization strategies for random forests are discussed in this section. In this position paper, we consider the implementation of random forests as single decision trees in the form of *if-else* trees, where tree nodes are compiled to *nested if and else* statements. Decision trees and random forests inherently follow a probabilistic model of how the data nodes of the tree are accessed during inference [2, 3]. This model naturally stems from the fact that the training data set is repeatedly split and assigned to a corresponding subtree. Consequently, the distribution of data elements in the split of the subtree can also serve as the probability during inference. This model is extracted during training. In a condensed representation, a relative probability of visiting the left or right child from each node is recorded. Towards this, we assume that each node $n \in N$ of a certain decision tree is either a leaf node or has two children nodes n_l and n_r. Additionally, each child has a probability of being accessed next during the inference $prob(n)$. Naturally, the probabilities of each pair of left and right children add up to 100% (i.e., $prob(n_l) + prob(n_r) = 100\%$), and the probability of the root node is also $prob(n_{root}) = 100\%$. Further, each node has a distinct access path starting from the root and ending in the node $path(n) = n_{root}, \ldots, n$. The absolute probability $prob_{abs}$ of a node to be accessed during a single inference run is the product of the probabilities of every node on the path to that node.

2.1 Register Caching

One approach to effectively make use of this probabilistic model during the implementation of a random forest model is to optimize the data storage of the decision tree in CPU registers. More concretely, the goal is to keep the most used data of the tree in the unused CPU registers to speed-up access to the values [1]. Towards this, it is assumed that the underlying CPU has a number of registers R. Additionally, a subset of $r < R$ of these registers can be deallocated for the code execution and manually used. Lastly, each of these registers is assumed to be capable of being written to and read from, as well as basic operations such as shifting the bits and comparing its content to other registers. These r registers are used to store the relevant values of the most frequently used nodes in the tree. As a realistic tree is too large to store all values in registers, a choice of which nodes to store needs to be made. We propose to order the nodes according to their absolute probability. The r most probable nodes are then chosen to be stored in registers. The relevant values are stored in the register in advance.

Since a single tree node naturally exceeds the size of a single register, a subset of the values to be stored per node needs to be chosen. The split value to compare against is always chosen, as it is always used for comparison. Additionally, depending on the concrete architecture, it can usually be placed in the bottom part of the register to be used for comparison immediately. If there is still further space in the register, other candidates to store are the indices to the left and right children, as well as the index to the feature value to compare against. Which to store can be chosen according to the needs of the current method. Due to the realization of if-else trees, methods of dynamically caching the feature values to compare against are further possible. During an inference run, some feature values might be used more than once. By keeping track of whether they were used in earlier nodes and which registers they were loaded into, these values can be used for comparison immediately. This is done while the code is being generated and, therefore, does not introduce any overhead during runtime.

2.2 Code Transformation

An orthogonal aspect of model agnostic optimization involves modifying the inference code through structural transformations and leveraging predication. Towards this, we propose two approaches of predication and branch reordering.

Predication: Most contemporary microarchitectures incorporate some form of branch prediction in their CPU designs. By predicting whether a given branch instruction is likely to be taken or not, control hazards can be mitigated, and costly pipeline flushes can be avoided. Leveraging the previously calculated (absolute) probability for each reached split in a decision tree $prob_{abs}$, we can implement the inference code in a branch-prediction-aware manner. If the absolute probability of a node is less than a certain threshold, branch predictors are expected to perform badly. When the remaining subtree in such a case is small, i.e., the depth crosses a certain threshold, subtrees can be implemented without branches by using predication. Predication is a technique that executes both branches of a conditional but conditionally retains only one, removing control hazards. Predication may outperform branched code if it is sufficiently small (thus, the overhead of executing additional instructions is negligible) and if branch prediction is suboptimal.

Branch Reordering: The second optimization strategy similarly capitalizes on the split probability $prob_{abs}$ to reorder selected branches with favourable probabilities. It ensures that the most probable samples require the fewest taken branches during execution. Considering a non-leaf node n with left and right children n_l and n_r, typical inference code generated during an if-else-tree implementation would entail a comparison of the node n's feature, followed by a branch instruction. In case the comparison holds true, the branch

instruction falls through and the directly adjacent code is executed. Conversely, if it evaluates to false, the branch is taken, requiring a jump to the next code block.

If the absolute probabilities of the child node $prob_{abs}(n_r)$ exceeds $prob_{abs}(n_l)$ by a certain factor $f_{reorder} \geq 0$, i.e., one outcome is significantly more probable, we rearrange the node n's children such that the new left child becomes the previous right child and vice versa. Simultaneously, we alter the comparison function to the complement (e.g., for $x \leq X$, we use $x > X$). This adjustment does not affect the decision tree's behavior, but ensures that high probable paths execute without the CPU having to take a jump. Similarly, the code generation of Treelite (i.e., this module is decoupled as TL2cgen from ver. 3.X) [6] realizes this concept by labelling the comparison condition through the compiler intrinsic `__builtin_expect`, which supplies the C compiler with branch prediction information. By utilizing a tuneable factor, we can search for the optimal reordering.

2.3 Quantization and Integerization

The data types, used by decision trees, i.e. the split values and the feature values, are decided by the type of the training data. This means that if the training data set consists of floating-point numbers, feature values and split values are naturally created as floating numbers as well. The usage of floating-point numbers can cause higher overheads in the execution time mainly due to two effects: 1) a direct effect of higher processing time for floating-point operations and 2) an indirect effect due to additional management overhead, such as copying to floating-point registers or additional memory loads instead of immediate encoding. Consequently, transforming floating-point-based decision trees to integer based decision trees is a favorable approach for performance optimization. While rounding the split values and possibly the feature values to integer equivalents (i.e. applying a form of quantization) is a straightforward solution (see, e.g., [8]), it induces a change in the model behavior and can affect the prediction quality. To overcome this, floating-point numbers, interpreted as bit vectors, can be reinterpreted as integer numbers, and integer operations can be used for the comparison. We ensure that the comparison operation of two given bit vectors under the interpretation as floating-point numbers is equivalent to the interpretation as integer numbers under a few conditions, which can be ensured at compile time [7]. This offers a form of integration, which does not affect the model behavior and the prediction quality.

3 Leveraging LLVM IR

The previously presented strategies for random forest optimization all operate on a conceptual level and can be implemented in a straightforward manner in dedicated implementations, which can be partially found in the literature [2, 5, 6]. Since the strategies, however, optimize for certain hardware properties, the implementations also are likely to become hardware-specific to a certain degree, e.g. implementations need to be pursued in assembly. An alternative approach is to consider LLVM Intermediate Representation (IR) to gain a sufficient control of hardware in a hardware-independent manner. This section envisions the realization of some of the aforementioned strategies in LLVM IR.

3.1 IR-Based Code Transformation

Different from straightforward approaches, LLVM allows for a higher-level abstraction beyond what assembly code can provide, while at the same time retaining low-level control. Creating the inference code directly in LLVM IR code achieves wide interoperability. The following is an illustration of how the above-mentioned tree code transformations of predication and reordering can be implemented in IR code: During code generation, two conditions are checked for each node n in a given decision tree T. These are ($reorder(n)$ and $branchless(n)$), as discussed in Section 2. They are evaluated to determine whether this node should be implemented with or without the corresponding optimizations.

By default, the generated IR resembles a **getelementptr** instruction followed by a **load** instruction to fetch the required feature in the feature array passed to the function. A **fcmp ugt** instruction then compares the fetched value with the split value of the node (embedded as a constant). Subsequently, the basic blocks for each of the left and right child nodes of the current node (if applicable) are recursively generated, such that they can be referenced in the current basic block via a **br** instruction. In case reordering should be performed, the **fcmp ugt** instruction in the above code is replaced with an **fcmp ule** instruction, and the basic blocks of the children are generated in the reversed order.

If a node is implemented in a branchless manner, after the comparison instruction, the IR code of both children is generated and inserted directly afterwards in the same basic block without a branch instruction. In this scenario, instead of the leaf nodes directly returning their prediction value, the current valid result is stored in a register. After the inference code of both children is executed. A **select** instruction is used to determine which of the children's results to retain. Due to its recursive nature, the implementation leads to the generation of various **select** instructions. The topmost node for which predication was utilized then returns (**ret**) the selected result.

3.2 Register Allocation

Investigating another advantage of LLVM IR puts a highlight on register allocation (RA). Since the number of physical registers is limited in the CPU, register allocation is performed as one step during compilation. That is, local variables and expression results are assigned to available registers automatically following a certain strategy. However, the strategy for

register allocation is designated for general-purpose applications, which may lead to inefficient spilling of registers for random forests and decision trees. Involving the knowledge about the probabilistic execution model of decision trees and random forests during the register allocation implements an instance of the previously explained register caching (Section 2.1) in a hardware-independent manner. To realize such an effective strategy in LLVM, e.g., by either adapting the default register allocation or designing a specific allocation for random forests, we envision that at least the following two issues in the default RA design should be considered.

First, the calculation of the live range for local variables and expression results by the default RA does not take the repeated data in the random forest into account, such that the registers are spilled inefficiently. When accessing the feature values, in the common implementations, pointers directed to data arrays are often used. However, these pointers, even if referencing the same data object, are calculated on demand and feature a short lifetime. Therefore, the repetitive access to feature values is not retained within the same register. We observe that many redundant procedures are automatically generated to dereference the same pointer again and again without holding in the registers. In order to tackle this issue, the lifetime of intermediate data to repeatedly access data elements has to be maximized.

Second, the scope of the default RA design is bound to the functions of the program. Typically, in a straightforward manner, each tree of a random forest is implemented as one function and gets called in sequence, prior to the majority voting for the final classification. If we directly apply the default RA with such an implementation of a random forest, the reused data among the trees is not properly handled in the calculation of live ranges. To overcome this issue, the implementation of random forests into single functions has to be re-evaluated and the lifetime of reusable data elements needs to be maximized.

3.3 In the Loop Profiling and Optimization

As a last approach, we envision the usage of LLVM IR for advanced profiling of the performance of optimization strategies. By utilizing LLVM IR code directly, the generated inference code can be statically compiled and linked to a suitable test environment that is previously compiled. This eliminates the need to rely on different compiler and linker infrastructures otherwise used for various programming languages, target architectures, and operating systems.

Employing performance application programming interface in conjunction with the aforementioned static compilation enables the creation of a fast, optimizable feedback loop. By incrementally adjusting the inference code affecting threshold values and strategies, the generated IR code is influenced. Utilizing a test environment that incorporates the widely used and platform-independent performance measurement library PAPI facilitates the creation of a target-unspecific testing framework. By leveraging the LLVM infrastructure, symbol conflicts between other compiler and linker frameworks can be entirely avoided. This enables rapid iteration of code generation required for extensive optimization loops.

4 Conclusion

In this paper, we envision model-agnostic optimization strategies for random forests. In greater detail, we approach optimization strategies, which operate on a trained random forest and do not affect the logic model behavior or the prediction quality. Since such approaches usually benefit from hardware-specific properties, they are naturally realized in a hardware-dependent manner. In order to surpass this, we envision the unified integration into the LLVM framework, which allows the application of hardware-specific optimization strategies in a general scope.

Acknowledgement This paper has been supported by German Research Foundation (DFG), as part of the project OneMemory(405422836), Memory Diplomat (502384507) and ARTS-NVM (502308721).

References

[1] Daniel Biebert, Christian Hakert, Kuan-Hsun Chen, and Jian-Jia Chen. 2024. Register Your Forests: Decision Tree Ensemble Optimization by Explicit CPU Register Allocation. *CoRR* abs/2401.15503 (2024).

[2] Sebastian Buschjager, Kuan-Hsun Chen, Jian-Jia Chen, and Katharina Morik. 2018. Realization of Random Forest for Real-Time Evaluation through Tree Framing. In *2018 IEEE International Conference on Data Mining (ICDM)*.

[3] Sebastian Buschjäger and Katharina Morik. 2018. Decision Tree and Random Forest Implementations for Fast Filtering of Sensor Data. *IEEE Transactions on Circuits and Systems I: Regular Papers* 65-I, 1 (2018).

[4] Sebastian Buschjäger and Katharina Morik. 2023. Joint leaf-refinement and ensemble pruning through L_1 regularization. 37, 3 (2023).

[5] Kuan-Hsun Chen, Chiahui Su, Christian Hakert, Sebastian Buschjäger, Chao-Lin Lee, Jenq-Kuen Lee, Katharina Morik, and Jian-Jia Chen. 2022. Efficient Realization of Decision Trees for Real-Time Inference. *ACM Trans. Embed. Comput. Syst.* 21, 6, Article 68 (2022), 26 pages.

[6] Hyunsu Cho and Mu Li. 2018. Treelite: toolbox for decision tree deployment. In *Proceedings of Machine Learning and Systems (MLSys)*.

[7] Christian Hakert, Kuan-Hsun Chen, and Jian-Jia Chen. 2022. FLInt: Exploiting Floating Point Enabled Integer Arithmetic for Efficient Random Forest Inference. *arXiv preprint arXiv:2209.04181* (2022).

[8] Simon Koschel, Sebastian Buschjäger, Claudio Lucchese, and Katharina Morik. 2023. Fast Inference of Tree Ensembles on ARM Devices. *CoRR* abs/2305.08579 (2023).

[9] Claudio Lucchese, Franco Maria Nardini, Salvatore Orlando, Raffaele Perego, Nicola Tonellotto, and Rossano Venturini. 2015. QuickScorer: A Fast Algorithm to Rank Documents with Additive Ensembles of Regression Trees. In *International ACM SIGIR Conference on Research and Development in Information Retrieval*. 73–82.

[10] Milan Shah, Reece Neff, Hancheng Wu, Marco Minutoli, Antonino Tumeo, and Michela Becchi. 2022. Accelerating Random Forest Classification on GPU and FPGA. In *Proceedings of the 51st International Conference on Parallel Processing, ICPP*.

accepted 2024-04-01