Efficient Inference with Accuracy-Preserved Integer Encoding on Tree-based Model Compiler

Jesper Lok
Computer Architecture for Embedded Systems
University of Twente, the Netherlands
Email: j.lok@student.utwente.nl

***Abstract*—In the ever expanding world of machine learning, where advancements continually push the boundaries of what's possible, the efficiency of deploying tree-based ensembles is often a point of interest. Renowned for their interpretability and minimal training data requirements, models such as random forests and extremely random forests have been found as effective tools across many different industries.**

In this paper, we integrate FLInt, a promising accuracy-preserving technique for accelerating inference speed, into the established TL2cgen framework. Through experimental analysis conducted on both x86 and ARM devices, we empirically demonstrate that FLInt consistently outperforms the baseline across a range of maximum tree depths. Additionally, we examine an established optimization technique, Quantization. However, our findings reveal that Quantization yielded mediocre results, often resulting in slowed inference times across most scenarios.

I. Introduction

Recent developments in machine learning have been made that focused on enhancing its capabilities and integration with popular machine learning frameworks. Specifically, tree-based models like decision trees and random forests are widely adopted tools for tasks across various industries, such as data science [1]. These models are explainable and require relatively little training data, which makes them particularly useful for applications in fields like finance, healthcare and national security [2].

Researchers have been exploring ways to improve the efficiency and performance of deploying tree ensemble models, addressing challenges such as reducing latency and optimizing resource utilization. Two well-known frameworks in this realm are TREEBEARD and Hummingbird. TREEBEARD for instance, uses new, advanced compilation techniques at various abstraction levels in order to lower inference times substantially [3]. On the other hand, Hummingbird aims to bridge the gap between traditional machine learning models and deep learning frameworks along with adding various other optimization to cut down on inference time [4]. A third framework that aims to contribute to the ongoing effort is an open-source library called "Treelite", and the now decoupled sub-module "TL2Cgen". Treelite lets users convert various well-known tree models into a common specification which is very useful when dealing with multiple applications that exchange and store decision trees. TL2Cgen can then use the Treelite model to generate highly optimized platform-independent C code, that can be compiled and run on any hardware. This is particularly useful for deploying tree models on devices where resources may be limited [5].

One technique that is used to improve efficiency is called "Quantization", which is commonly considered in other optimization problems such as binarized neural networks [6]. The problem with quantization is often the potential loss of accuracy. Furthermore, the overhead costs at prediction time may outweigh the gained latency reduction, thus making it counterproductive. The goal is to implement a novel method, called "FLInt", which is fully accuracy preserving, and see whether latency reduction can be achieved [7].

Our Contributions: In this project, we integrate FLInt into the existing TL2cgen framework, we evaluate inference times of decision trees from various datasets with different tree depths on two devices, and we answer the following research questions:

- What is the impact of optimization techniques, specifically Quantization and FLInt, on the inference times of random forest models generated by TL2cgen compared to the baseline scenario?
- Are there specific architectures or scenarios where FLInt or Quantization has superior performance compared to baseline?

The resulting code is publicly available at: https://github.com/Kolcenter/speedTest2

II. Related work

This section discusses works related to the topic of reducing inference times for tree-based machine learning models. First, an overview of TL2cgen and arch-forest, then we discuss an existing optimization technique in TL2cgen and lastly we review FLInt.

A. Deployment Frameworks

Besides TL2cgen, we already mentioned the TREEBEARD and Hummingbird framework, which both aim for performance increase in decision tree during inference. Another important framework to mention is "arch-forest", first developed by Buschjäger and Morik [8]. This framework has many overlapping features with TL2cgen and was the framework that was used to demonstrate FLInt's effectiveness. Some of the overlapping key features involve the following:

1) **C code generation:** Although greater optimization *can* be achieved by making platform-specific modifications

to decision trees, arch-forest and TL2cgen both aim to generate platform independent code. This is accomplished by creating the trees in C, which is widely used and relatively low-level because it provides direct access to memory addresses and hardware components, making it more efficient than some other languages like Python.

2) **if-else implementation:** The tree nodes, along with their respective values, can be stored in an arbitrary form. One way is to store the data as an array-like data structure and let a small loop read out node values while keeping track of the node's index, this is called a *native tree*. *if-else trees*, on the other hand, employ nested if-else blocks to represent nodes. When evaluating a branch condition, the code for the left subtree is enclosed within the if-block, and the code for the right subtree is within the else block. It has been empirically shown that an if-else implementation outperforms other structures, including the native implementation [9]. Hence, it is used in both arch-forest and TL2cgen.
3) **Cache-awareness:** Cache-awareness refers to the strategic utilization of computer memory hierarchy in software design to optimize performance. Computers generally have multiple types of memory with varying speeds and capacities, ranging from fast but limited cache memory to slower but larger main memory (RAM) and even slower storage devices like hard drives. In case of arch-forest, this feature is exploited by strategically placing child nodes within if-else statements, so that the more probable child is in the if-statement while the other child is in the else clause.
   In contrast, TL2cgen uses the built-in compiler feature `__builtin_expect()` to inform the compiler about the likelihood of different branches within the if-else statements. Both approaches essentially achieve the same thing; it ensures that the most frequently accessed parts of the decision tree are in faster (cache) memory.

B. Quantization

One optimization that is implemented in TL2cgen is referred to as Quantization. In this optimization, the test nodes undergo a transformation where all threshold values are substituted with integers, they are **quantized**. The result is that each threshold condition now involves integer comparisons instead of the usual floating-point comparisons. A simple example is shown in Listing 1

```
// Floating-point comparison
if (data[3].fvalue < 1.5) {
  ...
}
...
// Integer comparison
if (data[3].qvalue < 3) {
  ...
}
```

Listing 1. A simple example of a floating-point comparison (top) being quantized into an integer comparison (bottom)

As stated by the TL2cgen developer, this helps performance by reducing executable code size and improving data locality on some architectures, such as x86-64 [10]. This is due to the fact that, on these platforms, integer constants can be embedded as part of the comparison instruction, whereas floating-point constants cannot. Consequently, integer comparisons may produce fewer assembly instruction compared to floating-point comparison. However, using integer threshold will add overhead costs at prediction time, therefore it is not guaranteed that this optimization actually increases performance. Potential accuracy-loss is often a problem with quantization, but not in here since this quantization algorithm is fully accuracy-preserving.

C. FLInt

FLInt enables floating-point comparisons using only integers and logic operations [7]. In theory, this is useful particularly for devices lacking hardware floating-point units, such as small embedded systems, since those would require using software floating-points, which takes more time and energy. Other devices also typically experience execution time increase due to overhead introduced by floating-point operations. Theorem 1 shows the relation between a floating-point comparison and a signed integer comparison.

Theorem 1 (Restated from [7])**.** *Given two arbitrary bit vectors X, Y* $\in \{0,1\}^k$ *where the positiveness of FP(X) (equivalently SI(X)) is known a priori, the* $\geq$ *relation can be computed between the floating-point interpretation of these bit vectors, using only two's complement signed integer arithmetic:*

$$FP(X) \geq FP(Y)$$

$$\Leftrightarrow$$

$$\begin{cases} -1 \cdot SI(Y) \geq -1 \cdot SI(X) & \text{if } SI(X) < 0 \\ SI(X) \geq SI(Y) & \text{otherwise} \end{cases}$$

Here, $FP(X)$ and $SI(X)$ stand for the **F**loating **P**oint representation and the **S**igned **I**nteger representation of bit vector X respectively. Simply put, a computer can interpret the bit vector of a floating-point number as a signed integer and (with some extra steps in case the number is smaller than 0) the outcome of the comparison will be the same.
In [7], Hakert et al. have observed that FLInt integrated into the arch-forest framework results in an improvement of execution time for almost all evaluated cases. Considering the similarities between arch-forest and TL2cgen, it is only logical to expect FLInt to also have a positive impact on inference times for trees produced by TL2cgen.

III. Porting of FLInt into TL2cgen

The first step is to create a new branch for the package that will extend the capabilities of the current version to enable the user to pick FLInt as the applied optimization. Unlike Quantization, FLInt doesn't require manipulating thresholds and can directly be implemented at the section of code where the trees are generated. Listing 2 shows a simplified version

of the function `ExtractNumericalCondition`, which generates one if-statement for the decision tree based on the attributes of `node`.

```
std::string ExtractNumericalCondition(Node
    const* node) {
  std::string result;

  // Quantized threshold
  if (node->quantized_threshold_) {
    ...
    result = fmt::format("{lhs} {opname}
        {threshold}", ...
}
  // Regular threshold
  else {
    ...
    result= fmt::format("{lhs} {opname}
        ({threshold_type}){threshold}", ...
  }
  return result;
}
```

Listing 2. Simplified version of the ExtractNumericalCondition function in the TL2cgen source code

As can be seen, the code first checks whether the `quantized_threshold_` attribute of `node` is non-zero, and if it is, returns a string that contains the numerical condition of the decision tree for a quantized node. If `quantized_threshold_` is 0, the returned string is the numerical condition for a decision tree without optimizations (baseline). A similar approach can be taken for FLInt: add `FLInt_` as an attribute to `node`, which is set to 1 when the user wants to apply FLInt and 0 otherwise. Then, we edit the code shown in Listing 2 so that the attribute is checked, just like `quantized_treshold_`. Inside the if-statement, a numerical node for the decision tree is created based on theorem 1. A simplified version of the resulting code can be seen in Listing 3.

```
std::string ExtractNumericalCondition(Node
    const* node) {

  // Quantized threshold
  if (node->quantized_threshold_) {
    ...
    result = fmt::format("{lhs} {opname}
        {threshold}", ...
}
  // FLInt threshold
  else if (node->FLInt_) {
    ...
    result = fmt::format("{lhs} {opname}
        {threshold}", ...
  }
  // Regular threshold
  else {
    ...
    result= fmt::format("{lhs} {opname}
        ({threshold_type}){threshold}", ...
  }
  return result;
}
```

Listing 3. Simplified version of the ExtractNumericalCondition function after FLInt implementation

Creating a new decision tree with FLInt as the optimization is done in exactly the same way as one would with Quantization. Listing 4 shows the Python code that creates 3 decision trees: the baseline, with Quantization and with FLInt. These trees are stored at the specified directory path.

```
tl2cgen.generate_c_code(model,
dirpath=path_no_param, params={})

tl2cgen.generate_c_code(model,
dirpath=path_quantize, params={"quantize": 1})

tl2cgen.generate_c_code(model,
dirpath=path_FLInt, params={"FLInt" : 1})
```

Listing 4. Python code that generates 3 decision trees, The first with no parameters (baseline), the second with Quantization parameter applied, and the third one with FLInt parameter applied

Listing 5 shows a snippet of the decision trees from all 3 cases; baseline, Quantization and FLInt, respectively.

```
// Baseline
if (data[3].fvalue < (float)1.7999999523) {
  if (data[2].fvalue < (float)5) {
...
// Quantization
if (data[3].qvalue < 4) {
  if (data[2].qvalue < 6) {
...
// FLInt
if ((*( ((int*)(data)) + 3 )) <
    ((int)(0x3fe66666))){
  if ((*( ((int*)(data)) + 2 )) <
      ((int)(0x40a00000))) {
...
```

Listing 5. snippet of the decision trees from all 3 cases; baseline, Quantization and FLInt

Note that Quantization and FLInt only impact numeric splits and not categorical splits. This means inference times for categorical splits are unaffected by Quantization or FLInt.

IV. Evaluation

To evaluate the performance of the generated decision trees, we focus solely on measuring inference time, including possible overhead time costs. The general steps are straightforward: generate decision trees, then measure how long it takes to do a certain amount of inference tasks on those trees.

To do this, we create a program in Jupyter notebook because it allows for writing and executing code in small, manageable chunks, which is particularly useful when analysing data. The algorithm first creates all the decision trees that we are interested in testing. The program then creates random validation instances that are to be used on the decision trees. Some decision trees have more features than others, depending on the dataset, which means each dataset has to have a set of validation instances specific to that dataset. At last, the program generates a different C program that can load in the decision trees along with their respective validation set and test the inference times of the trees one by one. The times are collected by the C program and stored to a file to be used for further analysis.

A. Setup parameters

In order to comprehensively evaluate the speed of the generated decision trees across different scenarios, we explore various setup parameters. As mentioned in Section II, conditional thresholds may be handled differently based on the computer architecture. Hence, it is useful to evaluate different architectures. For this project, the evaluation is done on a desktop PC with an x86 architecture and on an ESP32-S3-DevKitC-1 running ARM. Because these devices have a significant difference in computational power, it is not suitable to create one test for both. Therefore, we have created two sets of parameters tailored to resemble real-life tasks in order to obtain a representative test for both devices.

It is important to emphasize that the goal of this evaluation is not to compare the hardware devices against each other, but rather to assess the performance of different optimizations with respect to each hardware platform. The parameters are as follows:

1) **Number of root nodes:** Less powerful devices such as embedded systems generally opt for decision trees with fewer root nodes in order to be computationally viable. This choice reduces the computational complexity of the tree, making it more feasible to deploy and execute on resource-constrained hardware. Therefore, the trees have *20 and 7 root nodes for x86 and ARM*, respectively.
2) **Maximum tree depths:** Different maximum tree depths will be investigated to assess their impact on inference time. By varying the maximum tree depth, we simulate varying levels of complexity encountered in real-world scenarios. The following maximum tree depths will be evaluated:
   (3, 5, 10, 15, 20) for x86 and (2, 3, 5, 8) for ARM
3) **Datasets:** The properties of a dataset can greatly impact inference times of its produced decision tree; for example the number of attributes. It is therefore important to use a diverse pack of datasets in order to get a good representation of the impact of an optimization on the inference time. Table I shows the list of datasets, along with the number of instances and the number of features. All eight sets will be used for the *x86* evaluation, while for *ARM*, datasets "Covertype", "Spam", and "Bankruptcy" are omitted. This is because these sets have relatively many features, which causes issues due to the hardware limitations of the ESP32-S3-DevKitC-1. These datasets are taken from the UC Irvine machine learning repository [11].

By systematically varying these parameters, we obtain an understanding of the speedups or slowdowns that are offered by Quantization or FLInt. This approach ensures that the findings are rigorous, plus relevant and applicable to practical applications.

B. Results

1) x86: Figure 1 shows the average speedup factors per dataset for baseline, FLInt and Quantization. FLInt is faster than baseline in all cases except for Spam ($\approx$ 0.5x). Quantization is slower than baseline in all cases, especially for Spam and Bankruptcy ($\approx$ 0.2x).

Dataset	# Instances	# Features
Breast cancer	569	30
Covertype	581012	54
Rice	3810	7
Wine	178	13
Magic	19020	10
Spam	4601	57
Glass	214	9
Bankruptcy	5306	95

TABLE I
DATASETS TAKEN FROM THE UCI DATABASE [11]

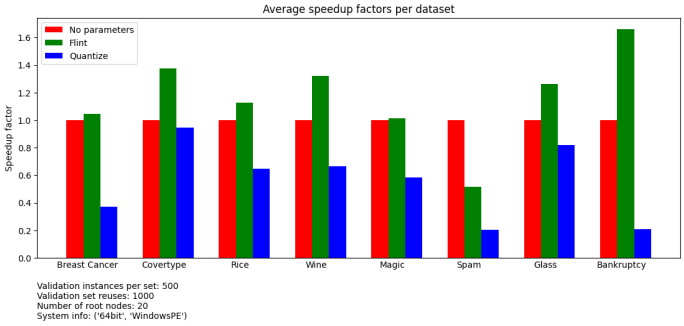

Fig. 1. Average speedup per dataset for baseline (red), FLInt (green) and Quantization (blue) with x86

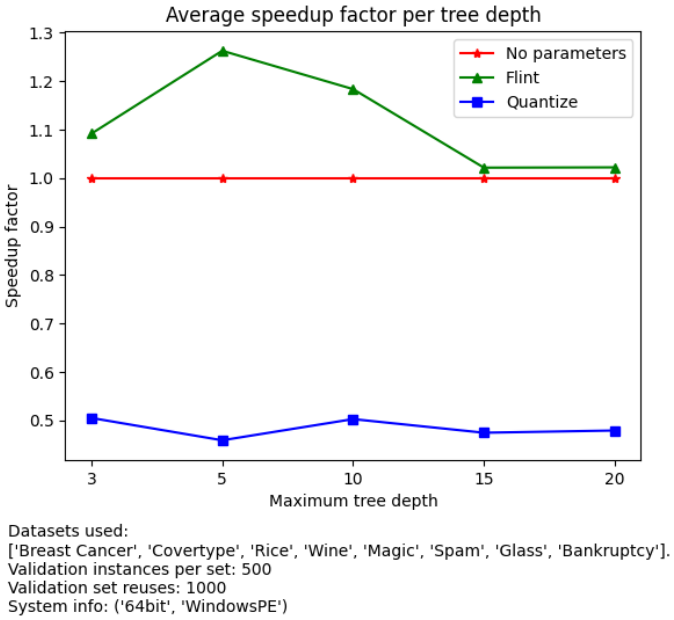

Fig. 2. Average speedup per tree depth for baseline (red), FLInt (green) and Quantization (blue) with x86

Figure 2 depicts the average speedup factors per maximum tree depth for all 3 cases. FLInt is faster and Quantization

is slower compared to baseline for all tested maximum tree depths. at tree depth 5, FLInt has the greatest speedup factor at around 1.25x and Quantization the lowest factor of around 0.45x. The speedup factor for FLInt gets closer to 1 as maximum tree depth increases after tree depth 5, while Quantization stays relatively constant at around 0.5x.

Overall, FLInt was faster than baseline in 32 out of 40 experiments and faster than Quantization in all tested cases, while Quantization was faster than baseline in 3 out of 40 instances (see fig. 6 in the appendix).

2) ARM: The speedup factor per dataset for baseline, FLInt and Quantization can be seen in Figure 3. FLInt outperforms baseline and Quantization for all datasets, especially for the ”Magic” and ”Rice” datasets (1.9x and 1.5x compared to baseline). These two averages are skewed by the results for tree depth = 8: As can be seen by fig. 7 in the appendix, the baseline time for ”Magic” dramatically increases between depths 5 and 8. The same is true for ”Rice”, though less extreme. Quantization outperforms baseline in three out of five datasets.

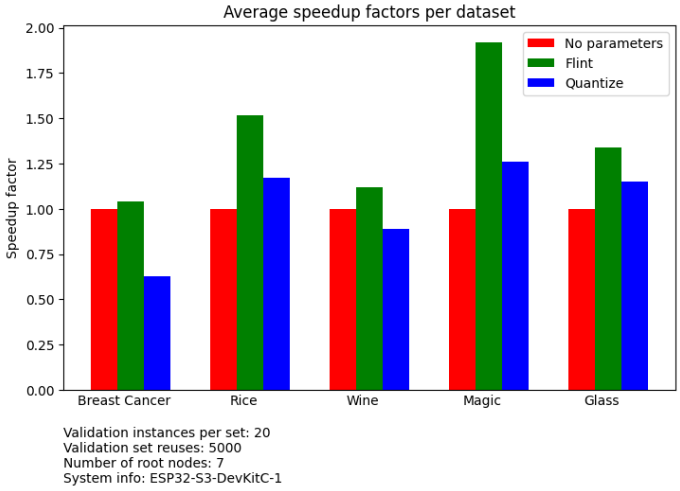

Fig. 3. Average speedup per dataset for baseline (red), FLInt (green) and Quantization (blue) with ARM

Figure 4 shows the average speedup factor per maximum tree depth for baseline, Quantization, and FLInt. Initially, Quantization is slower than baseline (maximum tree depth 3&5) but then slightly outperforms baseline at depth 5 and significantly at depth 8. On the other hand, FLInt outperforms the others in all cases, showing a similar curve as Quantization where the speedup factor gets larger as depth increases.

Furthermore, Figure 5 shows the average inference time per maximum tree depth. This plot shows that the increasing upwards slopes for Quantization and FLInt from Figure 4 are caused by the exponentially-appearing increase in inference time for the baseline, while Quantization and FLInt increase relatively linearly as the depth gets bigger.

Overall, FLInt was faster than baseline and Quantization in all 20 experiments, while Quantization was faster in 6 out of 20 instances (see fig. 7 in the appendix).

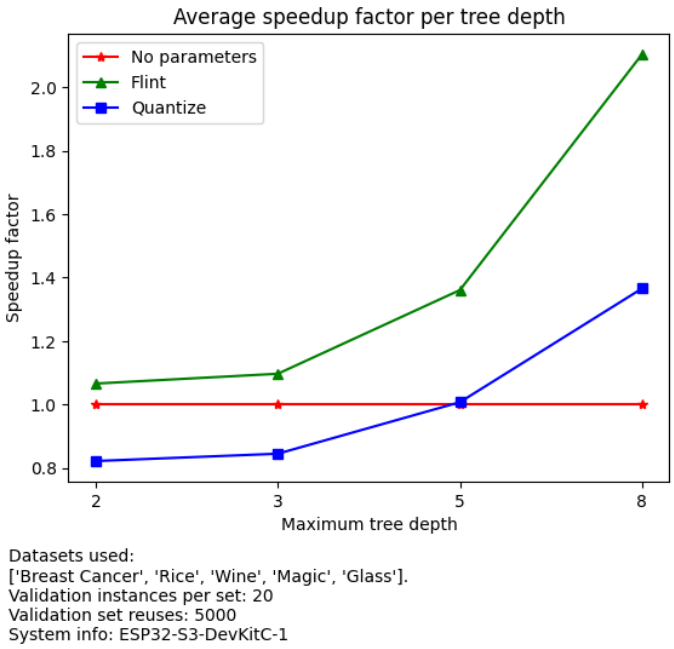

Fig. 4. Average speedup per tree depth for baseline (red), FLInt (green) and Quantization (blue) with ARM

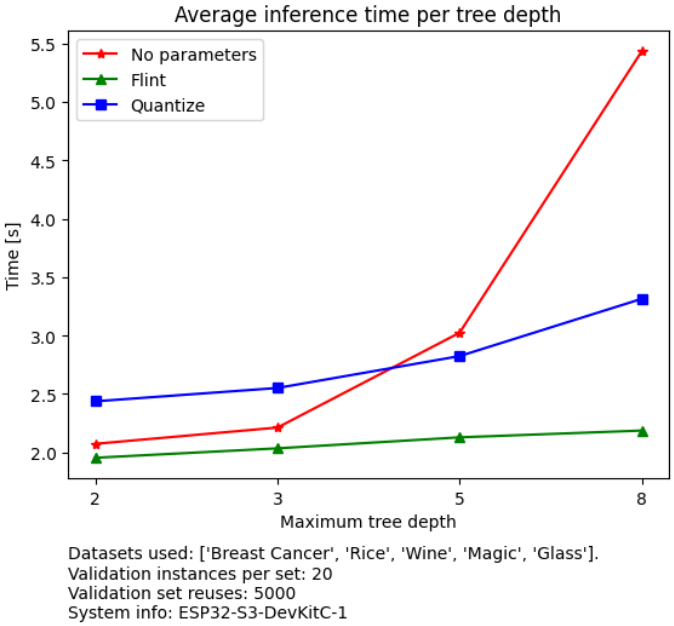

Fig. 5. Average inference time per tree depth for baseline (red), FLInt (green) and Quantization (blue) with ARM

C. Discussion

a) FLInt performance: Across architectures, FLInt consistently outperformed the baseline and Quantization, showcasing substantial speedup factors across datasets and tree depths. FLInt did especially well on the ARM device, where it had the lowest inference time for every experiment.

Cases where FLInt was only slightly faster may be caused

by a high number of categorical splits as opposed to numerical splits in a dataset, as those remain unaffected by FLInt. One might also think that numerical splits on features with integer values are unaffected by FLInt as well, but this is not true. Currently, TL2cgen doesn't generate different code for numerical splits with strictly integer values compared to floating-point splits.

Exploring FLInt's performance on a broader range of hardware, particularly on embedded systems lacking a floating-point unit may also give interesting insights. Unlike the ESP32-S3-DevKitC-1, many embedded systems do not feature a floating-point unit, relying on software-based floating-point operations that typically incur performance overhead, which could make FLInt even more effective on these devices.

b) Quantization: Quantization's performance was mixed at best, showing some promising results on ARM, but always falling short compared to FLInt. In extreme cases, quantize was almost 6.5x slower than baseline and over 8x slower than FLInt (see appendix, x86, bankruptcy3 and bankruptcy5-20, respectively). These slowdowns can likely be attributed to these factors: Firstly, the overhead introduced by the quantization process might be too substantial, negating any potential gains in inference speed. Secondly, while quantization aims to improve inference times by mapping threshold to integer values, the actual inference process may not have had a substantial speed improvement, leading to negligible latency reductions.

Further analysis could focus on identifying desirable dataset properties for the successful application of quantization. For example, Quantization may benefit from large trees, as in those cases, the relative time loss from the initial overhead might be outweighed by the gained inference latency reduction.

V. Conclusion

This project aimed to investigate the impact of optimization techniques, specifically Quantization and FLInt, on the inference times of random forest models generated by TL2cgen compared to the baseline scenario. Additionally, we were interested to see if there are specific architectures or scenarios where FLInt or Quantization have superior performance compared to the baseline.

In conclusion, our empirical analysis highlights FLInt as a promising addition to the TL2cgen toolbox. Our findings consistently demonstrate that FLInt outperforms both the baseline and Quantization in optimizing inference times. Although FLInt's effectiveness has been shown across a wide range of scenarios, it is not guaranteed to outperform the baseline in all of them. Conversely, Quantization has emerged as a rather mediocre optimization technique on the tested datasets, showcasing superiority over the baseline in only a handful of scenarios.

Further research is needed to understand which dataset or hardware properties are expected to benefit the most from FLInt and quantization, enabling more targeted optimization strategies in the future.

References

[1] Kaggle, "State of machine learning and data science 2020," 2020.

[2] D. Gunning, "Explainable artificial intelligence (xai)," 2017.

[3] A. Prasad, S. Rajendra, K. Rajan, R. Govindarajan, and U. Bondhugula, "Treebeard: An optimizing compiler for decision tree based ml inference," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022, pp. 494–511.

[4] A. Bahmani, Z. Xing, V. Krishnan, U. Ray, F. Mueller, A. Alavi, P. S. Tsao, M. P. Snyder, and C. Pan, "Hummingbird: efficient performance prediction for executing genomic applications in the cloud," *Bioinformatics*, vol. 37, no. 17, pp. 2537–2543, 03 2021. [Online]. Available: https://doi.org/10.1093/bioinformatics/btab161

[5] H. Cho and M. Li, "Treelite: toolbox for decision tree deployment," in *Proceedings of Machine Learning and Systems (MLSys)*, 2018.

[6] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks," *Advances in neural information processing systems*, vol. 29, 2016.

[7] C. Hakert, K.-H. Chen, and J.-J. Chen, "Flint: Exploiting floating point enabled integer arithmetic for efficient random forest inference," *arXiv preprint arXiv:2209.04181*, 2022.

[8] S. Buschjager and K. Morik, "Decision tree and random forest implementations for fast filtering of sensor data," 2017, pp. 19–28.

[9] S. Buschjager, K.-H. Chen, J.-J. Chen, and K. Morik, "Realization of random forest for real-time evaluation through tree framing," in *2018 IEEE International Conference on Data Mining (ICDM)*, 2018, pp. 19–28.

[10] "TL2cgen," accessed April 15, 2024. [Online]. Available: https://tl2cgen.readthedocs.io/en/latest/tutorials/optimize.html#use-integer-thresholds-for-conditions

[11] "UC Irvine Machine Learning Repository." [Online]. Available: http://archive.ics.uci.edu/

APPENDIX

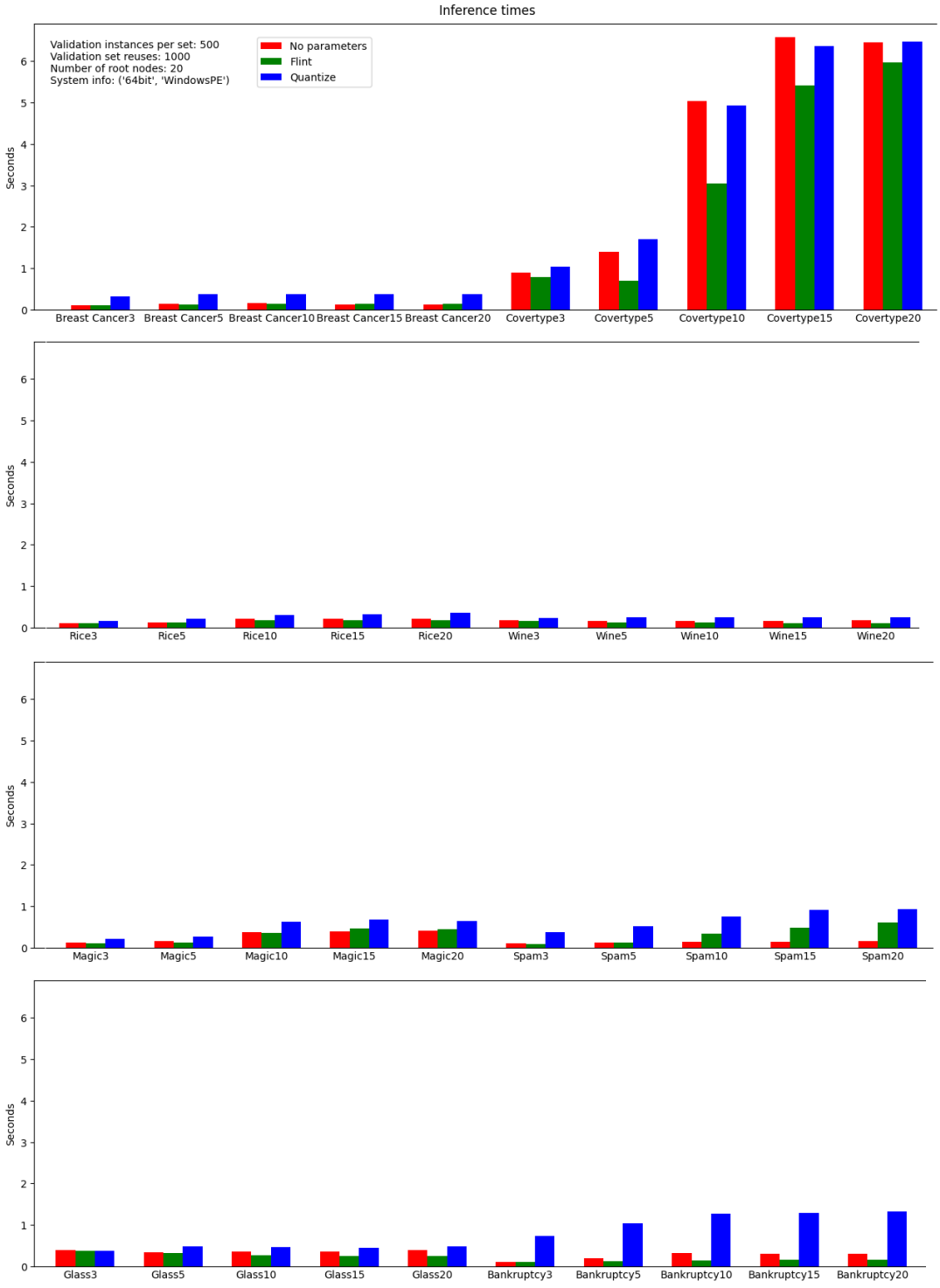

Fig. 6. Inference times for all experiments conducted on the x86 desktop PC

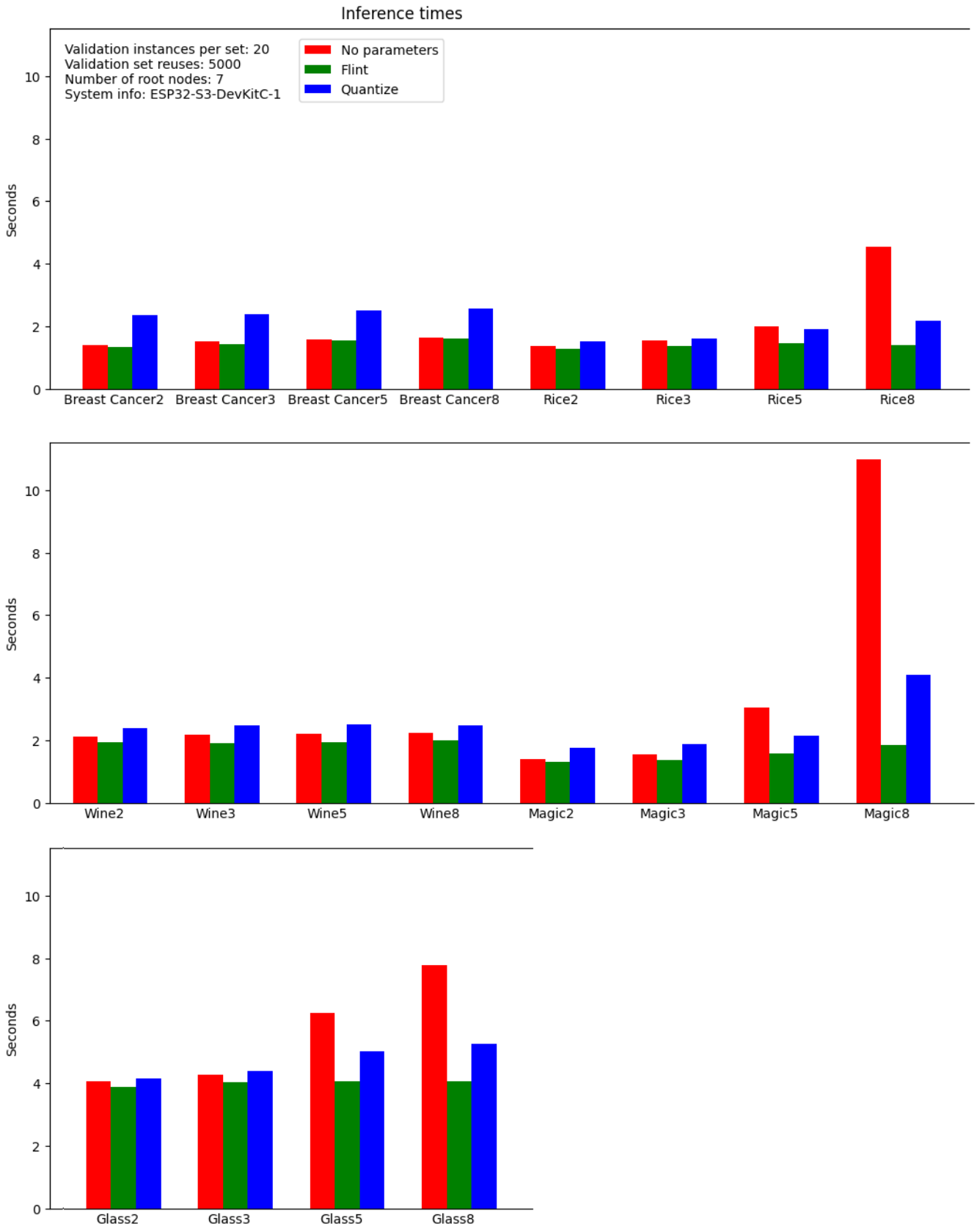

Fig. 7. Inference times for all experiments conducted on the ARM ESP32-S3-DevKitC-1