Biomedical Physics & Engineering Express

PAPER

High throughput software-based gradient tree boosting positioning for PET systems

Christian Wassermann 1,7,*, Florian Mueller 2,7,*, Thomas Dey 7,1, Janko Lambertus 7, David Schug 2,4,*, Volkmar Schulz 2,4,5,*,8, and Julian Miller 2,4,*,8

1 High Performance Computing Group, Computational Science and Engineering Division, IT Center, RWTH Aachen University, 52074 Aachen, Germany
2 Department of Physics of Molecular Imaging Systems, Institute for Experimental Molecular Imaging, RWTH Aachen University, 52074 Aachen, Germany
3 Faculty 05 Electrical Engineering and Information Technology, FH Aachen University of Applied Sciences, 52074 Aachen, Germany
4 Hyperion Hybrid Imaging Systems GmbH, 52074 Aachen, Germany
5 Physics Institute III B, RWTH Aachen University, 52074 Aachen, Germany
6 Fraunhofer Institute for Digital Medicine MEVIS, 52074 Aachen, Germany
7 Christian Wassermann and Florian Mueller contributed equally to this work and share the first authorship.
8 Volkmar Schulz and Julian Miller share the last authorship.
* Authors to whom any correspondence should be addressed.
E-mail: wassermann@itc.rwth-aachen.de and florian.mueller@pmi.rwth-aachen.de

Keywords: performance modeling, high-performance computing (HPC), performance optimization, singles and coincidence processing (SCP), positron emission tomography (PET), machine learning, gradient tree boosting (GTB)

Abstract

The supervised machine learning technique Gradient Tree Boosting (GTB) has shown good accuracy for position estimation of gamma interaction in PET crystals for bench-top experiments while its computational requirements can easily be adjusted. Transitioning to preclinical and clinical applications requires near real-time processing in the scale of full PET systems. In this work, a high throughput GTB-based singles positioning C++ implementation is proposed and a series of optimizations are evaluated regarding their effect on the achievable processing throughput. Moreover, the crucial feature and parameter selection for GTB is investigated for the segmented detectors of the Hyperion II\textsuperscript{3} PET insert with two main models and a range of GTB hyperparameters. The throughput model achieves singles positioning throughputs of more than 9.5 GB/s for smaller models and of 240 MB/s for more complex models on a recent Intel Skylake server. Detailed throughput analysis reveals the key performance limiting factors, and an empirical throughput model is derived to guide the GTB model selection process and scanner design decisions. The throughput model allows for throughput estimations with a mean absolute error (MAE) of 175.78 MB/s.

1. Introduction

Positron emission tomography (PET) is a functional, tracer-based, non-intrusive imaging technique with broad applications in both the clinical and preclinical domain (Myers 2001, Phelps 2004). During PET scans, a ring of radiation detectors registers two incoming gamma particles originating from an electron-positron annihilation, which occurs due to the radioactive $\beta^+$-decay of the used tracer. Scintillation crystals convert these gamma particles into optical photons, which are recorded by photosensor arrays. Based on the light pattern and timestamps registered by the photosensors, representing the raw sensor data, physical meaningful information about the gamma photon interacting needs to be derived. Typically, the spatial position and timestamp of the gamma photon interaction, as well as the amount of deposited energy in the scintillator, are calculated. This information is then used as an input for an image reconstruction algorithm.

In previous work, we presented a fully integrated, software-based data acquisition and processing architecture (DAPA) (Goldschmidt et al 2013, 2016). The data acquisition and processing server (DAPS) receives unprocessed raw photosensor data of all connected radiation detectors and either processes singles or coincidences in real-time or saves all data to drives for later offline processing. The software-based implementation
allows for high flexibility and reuse of existing software-based solutions. Furthermore, developed processing schemes are usually much simpler to port in a high-level programming language compared to hardware description languages as used for FPGAs.

The main processing steps of the current DAPS-implementation (see figure 1) include initial corrections of the raw photosensor data and clustering of independent photosensor channel information. Interaction position and deposited energy are estimated followed by the coincidence processing. This work focuses on the most time-consuming step of estimating the position of a single, which takes up around 50% of the total runtime. Additionally, the DAPS framework provides optional coincidence processing after the cluster process to reduce the subsequent workload, which was not used in this study.

Several data-driven modeling algorithms have been evaluated regarding their predictive capabilities to perform positioning of gamma interactions, especially in the domain of monolithic scintillator concepts (Lerche et al 2016, Gross-Weege et al 2016, Borghi et al 2016, van Dam et al 2011, Bruyndonckx et al 2008, Conde et al 2016, Wang et al 2013, Iborra et al 2019). Statistical models provide promising use-cases for detectors with segmented crystals as well, e.g., complex multi-layer staggered scintillators or local light-sharing approaches between single scintillator elements. Especially, the incorporation of depth of interaction (DOI) resolving capabilities may significantly increase the complexity of algorithms and processing times. When progressing from benchtop experiments to full PET rings and production systems with near real-time throughput requirements, the runtime remains as one main challenge for the wide application of statistical positioning algorithms.

Previous work has shown promising results for overcoming the runtime challenges while providing high spatial resolution through GTB methods (Müller et al 2018, 2019). GTB is a supervised machine learning technique that utilizes an ensemble of decision trees for classification or regression. While GTB predictors have been shown to require considerably less calibration data as opposed to other methods, GTB also scales well to three-dimensional positioning as its runtime only increases linearly with dimensionality (Müller et al 2018). For each positioning dimension, a separate GTB model is trained offline with a detector-specific calibration data set created by irradiating the detector at known positions using a dedicated coincidence collimator setup, e.g., a fan-beam collimator as described by Hetzel et al (2020). Furthermore, the GTB models can be configured via a set of hyperparameters, which control specific aspects of the training algorithm. A large space of parameter values allows for a trade-off between positioning accuracy and model complexity, which dictates computational requirements.

In this work, we demonstrate the real-time processing capabilities of GTB-based singles positioning algorithms based on offline raw data of the Hyperion II³ scanner (Weissler et al 2015). We extend the SCP framework (Goldschmidt et al 2013, 2016) to the GTB method presented in (Müller et al 2018, 2019) which acts as a drop-in replacement for the existing position estimation methods. Required GTB models were established based on a bench-top experiment with a single detector stack. The implementation is fully parallelized and optimized to handle online processing. To allow for an informed choice of suitable GTB parameter settings, we compare the processing throughput of different GTB model configurations and investigate the key factors which limit the processing throughput.
These findings generalize beyond the specific characteristics of the used scanner and computing architecture. To estimate the achievable throughput of a given GTB model for hardware procurement, resource allocation, and similar scanner design decisions, we derive an empirical throughput model.

## 2. Materials

### 2.1. Hyperion II²³ PET insert

As a detailed description of the Hyperion II²³ PET insert can be found in Weissler et al (2015), Hallen et al (2018), a short overview of the main components required for the data processing chain is given: The Hyperion II²³ insert consists of ten singles detection modules (SDMs) mounted circularly on an MR-compatible gantry. Every SDM hosts six detector stacks in a 2 × 3 arrangement as shown in Figure 2 and provides the infrastructure to operate the detector stacks, e.g., power distribution, liquid cooling, and RF shielding for operation inside the MR scanner. Every SDM is connected via User Datagram Protocol (UDP) to the DAPS employing an optical gigabit Ethernet link connecting and sending the raw data of all detector stacks. The Hyperion II²³ insert and the DAPS are controlled via a control PC connected to the DAPS. A separate synchronization unit distributes the reference clock and trigger signals.

Previous investigations (Goldschmidt et al 2016) found a maximum sustainable data rate of 95 MB/s per SDM after which data messages of the detector stacks are randomly discarded by the SDM and not sent out. Measurements with hot rod phantoms and cube sources revealed a nearly linear relation between activity and data input rate of 15.7 MB/(MBq s) to 19.1 MB/(MBq s). Depending on the configurations of the sensor tile, the detector stack itself saturates in the region of 40 MBq to 45 MBq. The specific configuration settings of the Hyperion II²³ PET insert as well as details of the used phantoms are presented in Goldschmidt et al (2016). These estimates lead to a maximum data rate of the Hyperion II²³ insert of around 860 MB/s until the detector stacks saturate and approximately 950 MB/s for the SDMs. Although the presented data rates may vary with different system configurations and are approximate numbers, they allow an informed assessment of the following data throughput rates.

#### 2.1.1. Detector stack

The detector stacks consist of a segmented scintillator array coupled via a lightguide to a sensor tile (see figure 2). The scintillator array is built up out of 30 × 30 optically isolated LYSO scintillator crystals with a pitch of 1 mm and 12 mm height. Optical photons are spread by the 2 mm thick lightguide to multiple read-out channels of the sensor tile. The sensor tile is custom-made variant made up of 16 digital SiPMs (DPC 3200-22, PDPC) with 2 × 2 read-out channels each resulting in 64 read-out channels and 16 time-to-digital converters, which generate the timestamps. Further information regarding the DPC 3200 are presented in Degenhardt et al (2009), Frach et al (2010).

Every DPC of the sensor tile acts as an independent unit following a two-level trigger scheme: After the first trigger condition, which logically connects 4 sub-regions of single read-out channels, is met, the DPC generates a timestamp and enters the validation phase of programmable length. The configurable validation criterion acts similarly to the trigger logic and is usually set to a higher level of required optical photons. Subsequent to a successful validation, the DPC starts the integration phase and, afterward, sums up all registered optical photons. Otherwise, the DPC resets and waits for the next trigger signal. After the integration phase, the photon counts of all four read-out channels of a DPC and their common timestamp are sent out.

#### 2.2. Single-detector calibration setup

Data sets for training and testing the GTB models with known irradiation positions were established using a bench-top coincidence calibration setup described and characterized in detail in Hetzel et al (2020). The main components of the setup are a fan-beam collimator with adjustable slit width hosting two $^{22}$Na

![Figure 2](image-url)
sources and an electrically driven translation stage defining the irradiation positions (LIMES 90, Owis, Staufen im Breisgau, Germany). Both sources had an activity of 5.5 MBq and the slit width was set to 0.4 mm for the calibration detector. The detector under test was placed on the translation stage and the coincidence detector on the opposing side. We utilized the PDPC technology evaluation kit (TEK) as a read-out infrastructure for the bench-top experiment. Therefore, we mounted an identical scintillator array and lightguide to the compatible PDPC DPC-3200 sensor tile. The whole setup was placed in a light-tight temperature chamber keeping the air temperature at constant 10 °C.

3. Methods

3.1. GTB-based positioning models

A detailed description of the GTB algorithm for predicting both planar and DOI positioning using monolithic scintillators is available in Müller et al (2018, 2019). As the method does not differ for this work, only the algorithm’s main characteristics and its hyperparameters used in this study will be summarized. The illustration in figure 3 outlines the central components of GTB-based positioning.

As part of supervised machine learning techniques, GTB builds a predictive positioning model based on a training data set with known irradiation positions. GTB models can act as classifiers or regressors. As previously shown, regression models reduce the number of known classes in the training data set, and, thus, the number of irradiation positions in a calibration process (Müller et al 2018). Therefore, only regression models are used in this work. The GTB algorithms accept incomplete data for training and evaluation as well as arbitrary input features (Chen and Guestrin 2016). During training, GTB builds a set of weak prediction models that consists of a series of binary decisions leading to a prediction. The second decision tree corrects the prediction of the first one. Two evaluation examples are shown employing the GTB model in (b). The prediction output is the sum of all predictions of the ensemble. Adapted from Müller et al (2018), CC 3.0.

As shown in previous works, the positioning accuracy of GTB models can be influenced significantly.
by adding information as further input features to the photon counts (Müller et al. 2018, 2019). This either improves positioning accuracy given the same hyperparameter settings or allows for smaller GTB models, which achieve the same positioning accuracy. For this work, the following two input feature sets were considered:

(i) The first feature set ($F_1$) only contains the recorded photon counts of each pixel. As the DPC and pixel with the highest number of counted photons are already calculated during previous steps of the DAPS processing pipeline, their indices were also added.

(ii) For the second feature set ($F_2$), the COG as well as the row- and column-wise photon count sums of the DPC pixels were appended to the feature vector.

While extra calculations are required for storing the additional feature values of $F_2$ models, these features carry more information than the photon counts of individual pixels in $F_1$.

### 3.1.1. Data generation and preprocessing

The calibration process was based on the single-detector calibration setup utilizing the fan beam collimator. We irradiated the scintillator array centrally at every line of the 30 scintillator segments, following the pitch of 1 mm. Before this measurement, the scintillator array’s edge was located by step-wise moving the detector into the beam as demonstrated in Ritzer et al. (2017). As mentioned earlier, the GTB algorithm was strictly used as a one-dimensional positioning approach. Thus, the radiation detector was rotated by 90° and the process was repeated.

The recorded raw data were preprocessed following the same scheme as shown in figure 1 employing an offline processing tool described in Schug et al. (2015). All hits of read-out DPCs were merged to clusters using a cluster window of 40 ns. The timestamp of the cluster was defined by the earliest timestamp of all related hits. Coincidence clusters were searched using a sliding coincidence window of 10 ns. No further quality cuts were applied to the data. As every DPC of the sensor tile is independent, not all available DPCs validate and sent out data. The clusters are arranged in vectors of 64 integers marking missing data as -1 to allow a distinction from zero photon counts.

### 3.1.2. GTB model generation

GTB models were trained using the XGBoost package (Chen and Guestrin 2016) which is integrated in a self-developed Python-based processing framework. Per irradiated line of scintillator segments, we selected 20000 gamma interactions for the training data set, leading to 600000 clusters in total for both x- and y-direction. The GTB models were validated and tested using validation and data sets containing in total 180000 cluster per spatial direction. The labels used for the GTB positioning models described the line of scintillator segments encoded by an integer number ranging from 0 to 30. The GTB regression models output a continuous estimation of the gamma interaction’s scintillator line. Thus, the output value was mathematically rounded in contrast to previous work. In case only integer values are used for the GTB evaluation within the DAPS framework (see section 3.3.4), the only continuous feature COG of feature set $F_2$ was multiplied by 100 prior to an integer cast retaining sufficient precision for these features. All trained GTB models were qualified by calculating their predictive performance using the test data set. We used the prediction accuracy, defined as the percentage of identifying the exact scintillator line corresponding to the irradiation position, as the evaluation metric. Due to Compton scatter inside the scintillator array, prediction accuracy of 1 is not achievable. Nevertheless, the prediction accuracy allows for consistent and easy comparisons between the positioning models.

### 3.2. GTB-based singles positioning

Focusing on the main singles processing task of position estimation, a new GTB-based singles positioning algorithm extends the existing DAPS software by integrating into its extendable class hierarchy. An abstract base class for all positioning algorithms holds necessary members for retrieving information about the used scanner arrangement and general configuration options common to all positioning algorithms. Appropriate child class methods for algorithm-specific options and actual position calculations are invoked via virtual function calls. The new GTB-based singles positioning (Müller et al. 2018, 2019) can therefore be chosen as an alternative to a COG calculation (Anger 1958) or a ML search (Lerche et al. 2016, Gross-Weege et al. 2016). Due to the framework’s modular design, the remaining parts of the processing chain do not require any modifications. Therefore, in the following, ‘throughput’ refers only to the throughput rate of the singles processing step as the other processing steps remained constant for all code versions.

As GTB positioning is performed by feeding feature vectors into pre-trained GTB models, the filenames of these models need to be specified in the application’s configuration file. For each detector stack and positioning dimension, a separate model is read in allowing for high flexibility in the chosen models and fine-grained control over the positioning accuracy. As described in section 3.1, feature set $F_2$ require calculations based on the measured photon counts. These calculations are performed within the GTB class to complete the input vector and are included in the throughput measurements.
3.3. GTB prediction implementations
The following sections present multiple runtime optimizations to accelerate GTB predictions. While the initial baseline utilizes library exposed application programming interface (API) functions to load GTB models into in-memory data structures at runtime, the subsequent modifications build on the idea of prior model compilation introduced by Asadi et al (2014). All of the introduced changes maintain the positioning accuracy of the pre-trained GTB models.

3.3.1. XGBoost online prediction (xgboost)
Since the GTB models were trained using XGBoost (Chen and Guestrin 2016), online prediction was integrated by invoking appropriate XGBoost C API functions. As no dedicated single instance prediction routine is exposed, the data structures need to be rebuilt repeatedly for each positioned single. To constraint the regression outputs for the x and y coordinates to valid crystal indices, the resulting GTB predictions need to be rounded and clipped accordingly. The associated code is shown in listing 1. Alternatively, the GTB models could be trained as classifiers to eliminate the need for post-processing the predictions.

3.3.2. Treelite model compilation (treelite)
To overcome the above API limitations and implement a technique presented in related research labeledCodeGen (Asadi et al 2014), the Treelite library for the compilation of tree-based machine learning models was used (Cho and Li 2018). Pre-trained GTB models are fed into a Treelite Model, which generates C source code consisting of a sequence of nested if-else-conditions.

After compilation to a shared object file, the model can be loaded during program initialization and used for later singles positioning via the exposed Treelite C API as displayed in listing 2. Compared to XGBoost, Treelite allows for single instance predictions and direct manipulation of feature values in the passed data structure, thus requiring fewer allocations and deallocations.

3.3.3. Primitive data type conversion (float)
In Treelite, feature values are implemented as unions. Missing features are indicated by setting the missing field to −1 and checking its value during prediction. These additional operations are eliminated by directly assigning a floating-point value of −1 to each missing photon count in the feature vector. This behavior has to remain consistent during training and prediction. As a consequence, the feature array only holds floating-point values and can directly be used for prediction. The implementation of this approach requires the transformation of the prediction interface. Instead of using Treelite’s functionality, the symbol of the predict function is looked up manually and called directly during prediction.

3.3.4. Float to integer quantization (int)
By converting floating-point feature values to an integer data type, prediction throughput may be increased further (Cho and Li 2018). On x86 architectures, integer comparisons are advantageous since constant values may be encoded into the operation directly. As an effect, the processor does not need to execute an additional load instruction to retrieve the integer constant. Ultimately, fewer instructions are required and the executable code size is reduced.

In general, this transformation entails overhead to guarantee lossless conversions without affecting the...
positioning accuracy of the GTB model. However, as input features for singles positioning only consist of discrete photon counts, the threshold in every branching condition may be replaced by its ceiling when using less-than comparisons. The only exception is the non-discrete COG coordinates of \( F_2 \), which were multiplied by 100 before the integer cast, to retain sufficient precision for these features.

3.3.5. Conditional branch annotation (annotated)
To further reduce the runtime required for the evaluation of the models, branch annotations may be introduced into the generated C source code before compilation. They provide additional information to the compiler regarding likely execution paths. Through branch reordering in the generated binary, data locality may be improved and branch predictions may become more accurate. This in turn improves instruction pipelining where incoming instructions are split into sequential steps. Fewer pipeline flushes occur which allows the waiting time for fetching memory due to memory access latencies to be overlapped with the execution of other instructions more efficiently.

Treelite’s built-in features are used (Cho and Li 2018) to generate annotations. The GTB model is evaluated on all training samples and executed paths are recorded individually for every branching condition. Conditions are marked as ‘likely’ if rendered true more than 50% of the time. The training data is assumed to approximate the online processed data with sufficient accuracy and, therefore, the provided annotations hold in the majority of cases.

3.4. Benchmarking methodology
For the evaluation of the implemented optimizations, a series of benchmarks were carried out on the RWTH compute cluster\(^9\). Each compute node is equipped with two Intel Xeon Platinum 8160 Skylake central processing units (CPUs) clocked at a base frequency of 2.1 GHz and 192 GB of main memory. The (CPUs) were allowed to increase their clock frequency if their temperatures allow for it (turbo boost). Each CPU features 24 physical cores while their hyper-threads were disabled. The cluster was running CentOS Linux release 7.6.1810 during the tests and the code was compiled with the Intel C/C++ Compiler 18.0.3 with the optimization options \(-g -03 -xHost\). The measurements cover increasing process counts scaling from a serial execution to a fully utilized compute node. To ensure reproducibility and reduce throughput variability, executions for each configuration were repeated 20 times with each process assigned to a specific core (process binding). The median of these measurements was used as a sensible figure of the expected throughput.

Previous research (Müller et al 2018, 2019) hinted towards reasonable parameter settings for achieving acceptable spatial resolution. Four GTB models ranging from low to high complexity due to their increasing number of trees and maximum tree depth were chosen to evaluate the impact of the implemented runtime optimizations on the throughput:

- tiny: \((\text{NumberOfTrees, MaximumTreeDepth, LearningRate}) = (20, 4, 0.2)\)
- small: \((\text{NumberOfTrees, MaximumTreeDepth, LearningRate}) = (100, 6, 0.2)\)
- medium: \((\text{NumberOfTrees, MaximumTreeDepth, LearningRate}) = (140, 8, 0.2)\)
- large: \((\text{NumberOfTrees, MaximumTreeDepth, LearningRate}) = (220, 10, 0.2)\)

By benchmarking the final code version across an extended parameter grid, the construction of a comprehensive throughput model for GTB positioning aims at throughput estimation for all GTB configurations. The following sets of values result in a total of 240 measured GTB models:

- Feature Sets: \(\{ F_1, F_2 \}\)
- Number of Trees: \([20, 60, 100, 140, 180, 220]\)
- Maximum Tree Depth: \([4, 6, 8, 10]\)
- Learning Rate: \([0.05, 0.1, 0.2, 0.4, 0.7]\)

3.5. Empirical throughput model
An empirical throughput model allows for an informed GTB model choice and guides the computing hardware procurement. A set of sensible parameters characterizing the achievable throughput of a given GTB model is presented and validated based on the findings of a low-level throughput analysis. These parameters can even be computed for previously untested GTB models. Therefore their throughput can be estimated without conducting runtime measurements.

Four main characteristics were considered to influence the parallel throughput of a GTB-based prediction model:

- The average number of evaluated conditionals per prediction can be estimated by evaluating the GTB model on its training data and counting the number of samples that end up in each specific
tree leaf. This distribution is aggregated into an average leaf depth and multiplied by the number of trees in the GTB model to get the desired figure (conditionals).

(ii) Memory related operations required for loading the binary instructions from the compiled GTB model are estimated by the GTB model’s byte size on disk (memory).

(iii) Modeling the exact behavior of hardware-based branch predictors would provide the most accurate branch misprediction ratio estimates. However, due to their growing complexity and dynamic behavior in modern processor architectures (Uzelac and Milenkovic 2009) this approach is infeasible in this context. Instead, a model’s branching behavior is represented by calculating how often the added compile time hint of the annotated version is incorrect by reusing the annotation data, which was generated for each conditional, and averaging it across the whole model (hint). The resulting value represents a penalty for non-contiguous code execution.

(iv) The number of executing processes (processes) is set during application execution and can, therefore, be used in the throughput model directly.

In the following, this modeling approach is referred to as the universal throughput model.

For an alternative throughput model, which only uses the initially supplied GTB model parameters, some of the above characteristics can be approximated or represented differently. The average number of evaluated conditionals per prediction is approximated by multiplying the maximum tree depth (depth) by the number of trees in the GTB model (trees). The GTB model’s binary size is approximated by multiplying 2^depth by trees. It is not possible to capture the branching behavior of GTB models only using its learning rate (learning) as this would not account for different input features. For this limited use case, we can indicate the used feature set \( F_1 \) or \( F_2 \) by setting features to 1 or 2 respectively. In the following, this modeling approach is referred to as the narrow throughput model.

Both throughput models were evaluated and compared. For both approaches, the benchmarking data described in section 3.4 was used to generate a linear least square regression model for the throughput of singles processing.

4. Results

For all but four of the collected measurements shown in figure 4, the 95% confidence interval fell within 10% of the reported medians such that the data was summarized in the following by taking the median throughput rate of all repetitions. The remaining four results showed no abnormal runtime behavior or outliers and were thus treated in the same manner. Percentages were calculated by taking the median percentages of all process counts, model sizes, feature sets, or code versions if appropriate.

4.1. Feature usage in GTB models

For all tested GTB parameter configurations, the \( F_2 \) models deliver higher positioning accuracies compared to the \( F_1 \) models with the same hyperparameter settings as indicated in figure 3. This is achieved through the additional features to which \( F_2 \) models have access during prediction namely the ‘cog’ and ‘sum’ data. The tiny \( F_2 \) model largely uses these calculated features to significantly increase its positioning accuracy above the \( F_1 \) model’s accuracy. With growing model size, the usage of the raw photon counts increases for both feature sets to allow for fine-grained adjustments to the estimated interaction position.

4.2. Throughput results

4.2.1. Throughput by model complexity

For all measured configurations, the achievable processing throughput declines with growing GTB model complexity. While a maximum throughput of 9.6 GB/s is possible when using the tiny model, the throughput peaks at 240 MB/s for the large model. The small, medium, and large models achieve 23.6%, 10.8%, and 3.2% of the tiny model’s throughput respectively.

When comparing the achievable throughput of an \( F_1 \) model to the \( F_2 \) model with the same hyperparameter setting, the collected data shows that in all cases the \( F_1 \) model can deliver a higher throughput. Overall, \( F_2 \) models achieve 81.3% of the throughput of \( F_1 \) models.

4.2.2. Throughput by code version

For all measured configurations, the achievable processing throughput increases when using compiled GTB models compared to the xgboost version. Exceptions occur when using the large model and more than 24 processes where the treelite and float versions fall below the xgboost version. The treelite version achieves 465% of the xgboost version’s throughput.

The float version achieves 108% of the treelite version’s throughput. The int version improves the throughput further as it achieves 135% of the float version’s throughput. While the annotated version only reaches 95.1% of the int version’s throughput at worst, in the median its throughput reaches 107% of the int version’s throughput.

4.2.3. Parallel scaling behavior

During parallel processing, all code versions achieve a sublinear median speedup—calculated as parallel
throughput divided by serial throughput—between 25.7 and 36.5 in case of a fully utilized compute node with 48 processes using the tiny, small, and medium models. This corresponds to a parallel efficiency calculated as speedup divided by number of used processes—between 0.54 and 0.76.

For the large model, the 	extit{xgboost} version achieves a perfect speedup in the majority of cases. The only exception is the large $\mathcal{F}_2$ model, for which an execution with 48 processes only achieves a parallel efficiency of 0.5. The highest throughput of all other code versions is achieved when partially utilizing the available processor cores by executing with fewer than 48 processes. The process threshold where throughput begins declining is specific to each code version. The earliest drops simultaneously occur for 	extit{treelite} and 	extit{float} which are followed by 	extit{int} and lastly 	extit{annotated}.

**Figure 4.** Singles processing throughput of the five presented code versions using four differently sized GTB models each with $\mathcal{F}_1$ and $\mathcal{F}_2$. In each subplot the same model is used for all code versions; thus their positioning accuracy is also identical.

**Figure 5.** The mean relative frequency of each feature group occurring in a split condition for four differently sized GTB models each with $\mathcal{F}_1$ and $\mathcal{F}_2$. The groups ‘raw’, ‘max’, ‘cog’, ‘sum’ contain the raw photon counts of each pixel, the max die and max pixel, the center of gravity, the row and column sums respectively. The frequencies are calculated by counting the split conditions which contain an index of the respective group divided by the number of total split conditions in the GTB model. The mean is derived by dividing by the number of individual indices in the respective feature group. The positioning accuracy of each GTB model configuration is indicated as ACC in the top left of each subplot.
4.3. Processing throughput analysis

4.3.1. Accessed memory levels

As the memory access latency varies between the different cache levels and main memory, the accessed level in the memory hierarchy has a substantial impact on the runtime of latency bound code regions. The results of the lat_mem_rd benchmark of the lmbench3 benchmark suite (McVoy and Staelin 1996, Staelin 2002) in table 1 show that latencies for these memory accesses can be reduced by a factor of 4 or 5 depending on the memory level.

This does not only impact serial runtime but also parallel executions. While the first and second cache levels are managed on a per-core basis, the last level cache is shared between all cores of the investigated CPU.

The visualization of the benchmarking data in figure 6 highlights the relationship between the memory level a compiled GTB model fits into and its achievable throughput. The four groups of curves correspond to the four different model sizes. While the tiny models fit into the L1 cache, the small and medium models fit into the L2 cache for all code versions and both feature sets respectively. However, for the large model all code versions and both feature sets eventually fill up the L3 cache at high process counts and models reside in main memory.

This shows that as long as one GTB model per executing process fits into at least the last level cache, throughput improves when adding more processes. Note that each executing process actually accesses two separate GTB model during each positioning calculation—one for each positioning dimension. However, since only parts of a GTB model need to be loaded depending on the taken branches, using the average size of just one GTB model per executing process leads to good approximations. When exceeding the described cache size threshold, achievable throughput rates may decline due to cache thrashing as the executing processes compete for the same limited cache storage when loading GTB models for prediction and, therefore, repeatedly need to access data from main memory. In these cases, the utilization of multiple threads per process enables the sharing of cache data, specifically of the GTB models. This could increase parallelism beyond the maximum number of processes and may lead to additional throughput improvements benefiting from the reuse of shared cache data. An evaluation of this proposition remains subject of future work.

4.3.2. Evaluating branch prediction

For tiny, small, and medium models, adding branch annotations only has a negligible impact on the achievable throughput. However, for large models, the annotated version improves the throughput beyond the optimizations of the int version. Measuring the branch misprediction shows that in case of the large $F_1$ model the misprediction ratio is reduced from 6.5% to 6.3% and in case of the large $F_2$ model the misprediction ratio is reduced from 7.3% to 6.6%. Consequently, the hardware prefetcher of the CPU can prefetch the correct data more frequently, which results in a measurable reduction of data traffic between the L2 and L3 cache. For the large $F_1$ model, the data movement is reduced from 16.6 TB to 11.5 TB while for the large $F_2$ model, it is reduced from 25.7 TB to 15.1 TB. Due to the improved accuracy of the prefetcher, memory access latencies can be hidden more frequently resulting in higher throughput. Therefore, the annotated version of

---

**Table 1.** Measured memory access latencies on a Skylake processor.

| Memory Level | Memory Size | Access Latency |
|--------------|-------------|----------------|
| L1 Cache     | 32 kB per Core | 1 ns           |
| L2 Cache     | 1024 kB per Core | 4 ns          |
| L3 Cache     | 34 MB per Socket | 20 ns         |
| Main Memory  | 192 GB per Node | 81 ns         |

**Figure 6.** Interaction between the memory consumption of compiled GTB models and the achieved processing throughput with one curve per code version, feature set and model size combination.
the models remains the preferred choice for further applications.

As the throughput of $F_1$ models surpasses the one of $F_2$ models with the same hyperparameter settings, their branch misprediction ratios can be compared as shown in table 2. Again, lower misprediction ratios coincide with higher throughputs.

### 4.3.3. Empirical throughput model

Due to its higher throughput, only the annotated implementation is used in the following.

The benchmarking data described in section 3.4 is used to generate a linear least square regression model for the throughput of singles processing. The formula for the universal throughput model is:

$$
\log(\text{throughput}) = +1.09 + \log(\text{processes}) - 0.04 \log(\text{processes})^2 + 1.67 \log(\text{memory}) - 0.07 \log(\text{memory})^2 - 0.42 \log(\text{hint}) + 0.19 \log(\text{hint})^2 - 1.02 \log(\text{conditionals}) - 0.35
$$

For the measured data, a mean absolute percentage error (MAPE) of 17.33% with a mean absolute error (MAE) of 175.78 MB is achieved.

As this regression model does not account for the throughput limitations imposed by the restricted L3 cache size, its accuracy degrades for GTB model configurations with a maximum depth of 10. Only considering GTB models which offer sufficient positioning accuracy and processing throughput, thus ignoring configurations with the largest and smallest tested parameter settings for the number of trees and the maximum tree depth, the MAPE and MAE of the same regression model are significantly lowered to 10.59% and 106.18 MB/s respectively.

Utilizing the previously described approximations for each term in the narrow throughput model leads to the following regression model, which uses a nested regression model for the memory footprint:

$$
\log(\text{throughput}) = +1.09 + \log(\text{processes}) - 0.04 \log(\text{processes})^2 + 0.64 \log(\text{memory}) - 0.04 \log(\text{memory})^2 + 0.06 \log(\text{learning}) - 0.10 \log(\text{learning})^2 - 0.22 \log(\text{features}) - 0.58 \log(\text{depth} \ast \text{trees}) + 6.25
$$

$$
\log(\text{memory}) = +0.11 \log(\text{depth} \ast \text{trees}) + 8.96
$$

For the measured data, a MAPE of 22.89% with a MAE of 201.20 MB/s is achieved.

### 5. Discussion

The presented implementation achieved data throughput rates between 240 MB/s and 9.6 GB/s depending on the GTB model for the investigated processing steps. When comparing these results to the throughputs of 12 350 MB/s and 3650 MB/s achievable by COG and ML positioning respectively on the same reference machine, this proves the feasibility of GTB-based real-time processing on the scale of PET systems considering the maximum data rate of Hyperion II\(^2\) insert of less than 1 GB/s. Furthermore, the DAPA allows the potential application of this technique for even larger PET systems while the presented insights help to match PET hardware and required processing architecture.

When selecting a specific GTB model for a given processing task, choices are mainly limited by two considerations: The desired positioning accuracy directly impacts the image quality, mainly in terms of the signal-to-noise ratio (SNR), and the required processing throughput to allow for real-time data processing. These two aspects are separately investigated in the following while the findings are combined afterwards to guide an informed GTB model choice.

As each process receives its own set of irradiation events, singles positioning is performed independently without the need for synchronization. However, the expected ideal scalability does not hold in all cases as the measurement results in figure 4 show. The irregular memory access pattern introduced by the chain of conditional jumps limits the runtime of the GTB prediction to the latency of the memory subsystem. Therefore, its runtime is mainly influenced by the accessed memory level and pipelining efficiency as detailed in the following.

As the memory access latency varies between the different cache levels and main memory (see table 2), the accessed level in the memory hierarchy has a substantial impact on the runtime of latency bound code regions and, thus, on the processing throughput of GTB models. When accessing the input feature vector, it will reside either in registers or the L1 cache due to its small size. However, while the compilation of the GTB model speeds up its execution significantly in the majority of cases, the resulting control flow dependencies introduce jumps through the dynamically loaded binary. Depending on the GTB model parameters and

| Model Size | $F_1$ | $F_2$ |
|------------|-------|-------|
| Tiny       | 7.4%  | 10.9% |
| Small      | 6.9%  | 8.9%  |
| Medium     | 7.0%  | 8.2%  |
| Large      | 6.3%  | 6.6%  |

Table 2. Branch misprediction ratios of all models with the annotated version.
the code version, its binary’s size can range from 20 kB to 7 MB.

The achieved throughput improvement from the float and int versions can be attributed to a smaller model binary size. While the prior eliminated one conditional check per if-condition, the latter optimization targeted specific limitations of the x86 instruction set thus further removing one load instruction per if-condition as described in section 3.3.4. As an effect, instructions are more often loaded from lower levels in the memory hierarchy.

To further improve the processing throughputs, one would need to lower the memory consumption of the GTB models beyond the already applied optimizations in this work. This includes the reduction of the number of distinct branching conditions through the union of similar subtrees (Nakamura and Sakurada 2019) and similar graph compression algorithms. However, these optimizations are typically lossy and may reduce the positioning accuracy.

The branch hints introduced by the annotated version only had a measurable effect on the runtime of the large model predictions. Due to improved branch predictions, the correct data is prefetched in more cases. Thus, memory access latencies are hidden more effectively, which directly improves the throughput of singles positioning. With increasing GTB model size, the access latency of the memory level in which a model can reside becomes more costly in turn growing the importance of this optimization.

Even if throughput requirements of a full PET system cannot be met by a single processing server, the flexibility of the DAPA allows the data stream of each singles processing unit (SPU) to be processed on independent machines. Only the final coincidence search requires aggregation into one coherent time line. Thus, processing throughputs can be increased beyond the results presented here.

While the universal throughput model uses the specific characteristics of a trained GTB model, its general formulation allows arbitrary GTB models to be evaluated. As an alternative throughput model that only uses the initially supplied GTB model parameters, the narrow throughput model could be preferred in order to avoid the potentially time-consuming training process of the GTB models. However, the applicability to GTB models with different feature sets is limited. Evaluating the two regression models on the measured data shows that the accuracy of the narrow throughput model lies significantly below the results of the universal throughput model.

While these throughput models offer throughput estimates on a specific processor and node configuration, the model could be generalized by incorporating characteristics like cache sizes to allow for its applicability to any underlying computing hardware. This remains subject of future work.

The selection of a GTB model represents a multi-objective optimization problem that aims to maximize the positioning accuracy of the chosen GTB model and its processing throughput. A set of optimal solutions can be generated by computing the Pareto frontier (Luc 2008). For these points, neither accuracy nor throughput can be improved without worsening the other criterion.

Applying this technique to the collected benchmarking data yields the 16 GTB model configurations in table 3 each with a given number of executing processes that serve as a baseline for the model selection. The visualization in figure 7 shows this same set of points as part of the complete benchmarking data. The vertical grouping of points stems from multiple executions with the same GTB model utilizing different amounts of parallel processes. Provided positioning accuracy estimates for a given GTB model parameter configuration, the same methodology can be combined with either of the above throughput models to

| #Procs | Feat Set | #Trees | Depth | LRate | Accuracy | Throughput |
|--------|----------|--------|-------|-------|----------|------------|
| 48     | 1        | 20     | 4     | 0.1   | 0.133    | 10 860     |
| 48     | 1        | 20     | 4     | 0.7   | 0.342    | 9838       |
| 48     | 2        | 20     | 4     | 0.7   | 0.576    | 7852       |
| 48     | 2        | 20     | 6     | 0.7   | 0.599    | 6564       |
| 48     | 2        | 20     | 6     | 0.4   | 0.605    | 6018       |
| 48     | 2        | 20     | 8     | 0.7   | 0.612    | 5368       |
| 48     | 2        | 20     | 8     | 0.4   | 0.619    | 4421       |
| 48     | 2        | 20     | 10    | 0.7   | 0.621    | 4272       |
| 48     | 2        | 20     | 10    | 0.4   | 0.630    | 3131       |
| 48     | 2        | 60     | 10    | 0.4   | 0.635    | 1768       |
| 48     | 2        | 100    | 10    | 0.4   | 0.635    | 1142       |
| 48     | 2        | 60     | 10    | 0.2   | 0.636    | 1075       |
| 48     | 2        | 180    | 8     | 0.2   | 0.636    | 770        |
| 48     | 2        | 220    | 8     | 0.2   | 0.636    | 645        |
| 48     | 2        | 100    | 10    | 0.2   | 0.638    | 596        |
| 48     | 2        | 140    | 10    | 0.2   | 0.639    | 354        |
generate analogous insights without requiring extensive runtime measurements.

Examining the trends in the Pareto frontier data reveals that in order to increase positioning accuracy and to retain optimality, first the additional calculated features should be added to a GTB model. Although $F_1$ models achieve higher throughputs for the same hyperparameter settings compared to $F_2$ models, $F_2$ models should almost always be preferred over $F_1$ models when trying to achieve the same positioning accuracy. Next, it is most beneficial to increase the tree depth even up to depth 10. Further trees should be added last as it improves the positioning accuracy only marginally. The optimal learning rate decreases from 0.7 to 0.4 when trees are grown deeper. Here, a value of 0.2 should only be chosen in large models for minor positioning accuracy gains.

6. Conclusion

This work demonstrates the real-world applicability of GTB-based singles positioning by providing a high throughput implementation in a parallelized processing framework for PET SCP. A range of optimizations including model compilation and data type conversion was shown to significantly increase the processing throughput resulting in an achievable throughput of more than 9.5 GB/s. A throughput analysis revealed that GTB prediction is mainly limited by memory access latencies due to irregular memory accesses of the compiled GTB model binary. Thus, reducing the memory footprint of these binaries such that they fit into lower-level caches was shown to be beneficial for increasing throughput. Additionally, branch annotations proved effective in improving the pipelining efficiency and hiding memory access latencies by lowering branch misprediction ratios. For the fully optimized implementation, a precise throughput model was derived that enables throughput estimation with a MAPE of 17.33% and a MAE of 175.78 MB/s.

In the future, the processing throughput of GTB-based singles positioning could be further improved by lowering the memory consumption via compression of GTB models (Nakamura and Sakurada 2019). Similarly, GTB models could be split into a range of trees that fit into a specific cache level and evaluating this tree subset on all instances before progressing to the next range of trees (Ilic and Kuvshynov 2017). Moreover, previous research (Asadi et al. 2014) suggests converting control dependencies occurring due to branching into data dependencies by employing predication. Thus, jumps in the underlying assembly code are replaced by memory accesses which also cause memory access latencies. However, vectorization may hide these latencies and speed up prediction overall. The presented throughput model may be generalized to become architecture-independent by incorporating specific cache sizes and different memory access latencies within the memory subsystem. This would enable more accurate throughput predictions when progressing to new computing hardware.

Acknowledgments

This project has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No. 667 211. Funded by the Federal Ministry of Education and Research (BMBF) and the Ministry of Culture and Science of the German State of North Rhine-Westphalia (MKW) under the Excellence Strategy of the Federal Government and the Länder under grant number G: (DE-82)EXS-PF-JARA-SDS008.

Data availability statement

The data that support the findings of this study are available upon reasonable request from the authors.
ORCID iDs

Christian Wassermann  https://orcid.org/0000-0002-5044-6137
Florian Mueller  https://orcid.org/0000-0002-9496-4710
David Schug  https://orcid.org/0000-0002-5154-8303
Volkmar Schulz  https://orcid.org/0000-0003-1341-9356
Julian Miller  https://orcid.org/0000-0001-5564-0775

References

Anger H 1958 Scintillation camera Rev. Sci. Instrum. 29 27–33
Asadi N, Lin J and de Vries A P 2014 Runtime optimizations for tree-based machine learning models IEEE Trans. Knowl. Data Eng. 26 9228–92
Borgi G, Peet B J, Tabacchini V and Schaart D R 2016 A 32 mm × 32 mm × 32 mm monolithic LYSO:Ce detector with dual-sided digital photon counter readout for ultrahigh-performance TOF-PET and TOP-PET/MRI Phys. Med. Biol. 61 4929–49
Bruyndonckx P, Lemaitre C, van der Laan D J, Maas M, Schaart D, Borghi G, Peet B J, Tabacchini V and Schaart D R 2016 A likelihood positioning algorithm for high-resolution PET detector raw data IEEE Trans. Nucl. Sci. 63.2 669–76
Bruyndonckx P, Lemaitre C, van der Laan D J, Maas M, Schaart D, Borghi G, Peet B J, Tabacchini V and Schaart D R 2016 A likelihood positioning algorithm for high-resolution PET detector raw data IEEE Trans. Nucl. Sci. 63.2 669–76
Chen T and Guestrin C 2016 XGBoost: a scalable tree boosting system Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD ’16 (New York, NY: ACM) 785–94
Cho H and Li M 2018 Treetoric: library for decision tree deployment The Conference on Systems and Machine Learning (Stanford, CA: SysML’18) https://mlsys.org/Conference/docs/2018/196.pdf
Conde P et al 2016 Determination of the interaction position of gamma photons in monolithic scintillators using neural network fitting IEEE Trans. Nucl. Sci. 63 130–6
van Dam H T, Seifert S, Vinke R, Dendooven P, Lohner H, Bekeman F J and Schaart D R 2011 Improved nearest neighbor methods for gamma photon interaction position determination in monolithic scintillator PET detectors IEEE Trans. Nucl. Sci. 58.5 2193–47
Degenhardt C, Prescher G, Fracht T, Thon A, de Gruyter R, Schumacher A and Ballanzy R 2009 The digital Silicon Photomultiplier — A novel sensor for the detection of scintillation light 2009 IEEE Nucl. Sci. Symp. Conf. Rec. (Piscataway, NJ: IEEE) 2383–6
Fracht T, Prescher G, Degenhardt C and Zwaans B 2010 The digital silicon photomultiplier — System architecture and performance evaluation IEEE Nucl. Sci. Symp. Med. Imaging Conf. 1722–7
Goldschmidt B, Lerche C W, Solf T, Salomon A, Kiesling F and Schulz V 2015 Towards software-based real-time singles and coincidence processing of digital PET detector raw data IEEE Trans. Nucl. Sci. 60.3 1593–9
Goldschmidt B, Schug D, Lerche C W, Salomon A, Gebhardt P, Weissler B, Wehner J, Dueneppenbercker P M, Kiessling F and Schulz V 2016 Software-based real-time acquisition and processing of PET detector raw data IEEE Trans. Biomed. Eng. 63.2 2316–27
Gross-Wügel N, Schug D, Hallen P and Schulz V 2016 Maximum likelihood positioning algorithm for high-resolution PET scanners Med. Phys. 43 3049–61
Hallen P, Schug D, Weissler B, Gebhardt P, Salomon A, Kiesling F and Schulz V 2018 PET performance evaluation of the small-animal hyperion II D PET/MRI insert based on the NEMA NU-4 standard Tech. Rep. 4063027
Hetzel R, Mueller F, Graje H, Honné A, Schug D and Schulz V 2020 Characterization and simulation of an adaptable fan-beam collimator for fast calibration of radiation detectors for PET IEEE Transactions on Radiation and Plasma Medical Sciences 4.5 538–45
Iborra A, Gonzalez Martinez A J, Gonzalez-Montoro A, Bousse A and Visvikis D 2019 Ensemble of neural networks for 3D position estimation in monolithic PET detectors Phys. Med. Biol. 127.4 549–59
Kotsiantis S 2013 Decision trees: A recent overview Artif. Intell. Rev. 39.4 261–83
Lerche C, Salomon A, Goldschmidt B, Lodomez S, Weissler B and Solf T 2016 Maximum likelihood positioning and energy correction for scintillation detectors Phys. Med. Biol. 61 1650–78
Luc D 2008 The Pareto Optimality. In: Pareto Optimality, Game Theory And Equilibria ed A Chinchuluun et al (New York, NY: Springer) pp 481–315
McVoy L and Staelin C 1996 Lmbench: Portable Tools for Performance Analysis USENIX 1996 Annual Technical Conference 279–94 https://www.usenix.org/legacy/publications/proceedings/sd96/full_papers/mcvoy.pdf
Muller F, Schug D, Hallen P, Graje H and Schulz V 2018 Gradient tree boosting-based positioning method for monolithic scintillator crystals in positron emission tomography IEEE Transactions on Radiation and Plasma Medical Sciences 2.3 411–21
Muller F, Schug D, Hallen P, Graje H and Schulz V 2019 A novel DOI positioning algorithm for monolithic scintillator crystals in PET based on gradient tree boosting IEEE Transactions on Radiation and Plasma Medical Sciences 3.4 465–74
Myers R 2001 The biological application of small animal PET imaging Nucl. Med. Biol. 28.5 585–93
Nakamura A and Sakurada K 2020 An Algorithm for Reducing the Number of Distinct Branching Conditions in a Decision Forest Joint European Conference on Machine Learning and Knowledge Discovery in Databases 578–89
Natekin A, Knoll A, Gewaltig M-O and Michel O 2013 Gradient boosting machines, a tutorial Frontiers in Neurorobotics 7 21
Pheils M E 2004 PET: Molecular Imaging and Its Biological Applications (New York, NY: Springer) https://doi.org/10.1007/978-0-387-22529-6
Ritzger C, Hallen P, Schug D and Schulz V 2017 Intercrystal scatter rejection for pixelated PET detectors IEEE Trans. Radiat. Plasma Med. Sci. 1.2 191–200
Schug D, Wehner J, Goldschmidt B, Lerche C W, Duppenenbercker P M, Hallen P, Weissler B, Gebhardt P, Kiesling F and Schulz V 2015 Data processing for a high resolution preclinical PET detector based on phils DPC digital SiPMs Nucl. Sci. IEEE Trans. 62.3 669–78
Staelin C 2002 Lmbench3: measuring scalability Technical report, HP Laboratories Israel https://www.hpl.hp.com/techreports/2002/HPL-2002-313.pdf
Uzelac V and Milenkovic A 2009 Experiment Flows and Microbenchmarks for Reverse Engineering of Branch Predictor Structures 2009 IEEE International Symposium on Performance Analysis of Systems and Software 207–17
Wang Y, Zhu W, Cheng X and Li D 2013 3D position estimation using an artificial neural network for a continuous scintillator PET detector Phys. Med. Biol. 58.5 1375–90
Weissler B et al 2015 A digital preclinical PET/MRI insert and initial results IEEE Trans. Med. Imaging 34.11 1–1