Hardware Support for Intelligent Text Analysis using FPGA for Accelerating Random Forest-based Classification

Vishniakou Uladzimir Anatol'evich
Belarusian State University of Informatics and Radioelectronics
Minsk, Belarus

Yu ChuYue
Belarusian State University of Informatics and Radioelectronics
Minsk, Belarus

ABSTRACT
Efficient analysis and classification of text performed at the edge of a network, especially on platforms with limited resources such as embedded systems and FPGA devices, creates computational challenges. Traditional CPU and GPU-based natural language processing (NLP) methods struggle to meet the real-time and energy efficiency requirements of peripheral computing scenarios. To eliminate these limitations, this study suggests hardware support for an FPGA-based random forest algorithm for text classification. To meet the resource constraints inherent in embedded and FPGA-based systems, the proposed methodology includes model compression, simplified algorithmic optimization, fixed-parameter configurations, fixed-point computing, and dimensionality reduction techniques, which effectively reduces both computational complexity and memory consumption. A hybrid CPU-FPGA pipelining architecture has been developed, in which the central processor performs text preprocessing tasks, including tokenization, TF-IDF vector computing, and function normalization, while the FPGA accelerates data output from the random forest algorithm using parallel computing and pipelining strategies. The FPGA implementation has been thoroughly tested for compliance with the Python-based reference processor model through a joint software and hardware verification process. The results demonstrated a high degree of numerical consistency, reaching a similarity of 0.9990, which confirms the correctness of the end-to-end logic of feature extraction and classification. The proposed FPGA architecture provides a scalable solution for high-performance, low-latency NLP applications suitable for deployment in peripheral computing environments.

General Terms
Algorithms, Hardware Acceleration, Natural Language Processing, FPGA, Embedded Systems, Edge Computing, Machine Learning, Performance Optimization, Verification.

Keywords
FPGA, Random Forest, Text Analytics, TF-IDF, Hardware Acceleration.

1. INTRODUCTION
Text analysis techniques are commonly required to efficiently process massive amounts of textual data, extract key information, and classify it in application scenarios such as medicine, public opinion monitoring, and customer service. For instance, NLP techniques can extract meaningful information from unstructured clinical text data, including electronic health records, medical literature, physician notes, and patient reports [1]. The most commonly used BERT models in biomedical text processing include BioBERT, SciBERT, and PubMedBERT [2].

Traditional natural language processing (NLP) tasks typically rely on CPU or GPU computing, which is often computationally intensive and energy-consuming. In edge computing scenarios that require low latency and high energy efficiency, these environments generally have limited computational resources [3], making it difficult to meet the needs of real-time processing or low-power applications. Edge computing has applications in autonomous vehicles, smart cities, and healthcare. To meet these demands, the following strategies can be adopted: model compression, lightweight algorithms, edge-cloud collaboration, and the use of specialized hardware accelerators such as GPUs and FPGAs to enhance processing efficiency. Model compression involves techniques such as quantization, pruning, and knowledge distillation to reduce model size, although some accuracy loss may occur [4]. Lightweight algorithms aim to reduce code size and memory consumption by employing resource-efficient algorithms [5]. Edge-cloud collaboration involves offloading resource-intensive tasks to the cloud to alleviate the burden on edge devices [6]. Specialized hardware accelerators refer to the use of GPUs and FPGAs to improve processing efficiency [7]. These solutions facilitate efficient text processing and real-time response in resource-constrained environments.

The application of FPGAs (Field-Programmable Gate Arrays) in machine learning acceleration has become increasingly widespread. However, applying FPGA hardware acceleration to NLP tasks requires addressing challenges such as efficiently handling large-scale vocabulary lookups, text preprocessing, and integrating traditional software components with hardware modules. In machine learning algorithms, CNNs [8] and deep learning models such as BERT [9] have been deployed on FPGAs, demonstrating improved throughput and energy efficiency compared to pure CPU-based solutions. Models such as SVM and BERT can be directly compressed through quantization and pruning, whereas random forests require model simplification (e.g., compact random forests, CRFs) to accommodate FPGA resource constraints. Additionally, TF-IDF feature extraction involves dynamic memory allocation (e.g., term frequency statistics and inverse document frequency computation) and extensive string operations, making these tasks difficult to directly map onto FPGA hardware using hardware description languages (such as Verilog) or high-level synthesis (HLS) tools. Despite the existing gap between CPU-based NLP feature extraction and FPGA-based classifiers, hybrid architectures provide an optimized approach for NLP processing. This study aims to design an FPGA-accelerated text classification architecture based on random forests to

achieve high-performance, low-latency intelligent text analysis applications in resource-constrained environments.

2. METHODOLOGY

To achieve efficient FPGA deployment, the overall research design consists of five stages: data collection and preprocessing, software-based model training and lightweight optimization, FPGA hardware design and adaptation, verification and performance evaluation, and engineering considerations for edge computing deployment. Specifically, after acquiring the dataset, training the model, and verifying its performance, the adopted algorithms undergo lightweight optimization and hardware-adaptive modifications to ensure that the model size and data structures match FPGA resources and characteristics. Subsequently, the core algorithm modules are implemented in fixed-point C++ to achieve superior performance and resource utilization efficiency on hardware. Finally, the design is subjected to hardware behavioral simulation using HLS tools to verify functionality and preliminary performance, considering additional factors present in actual deployment scenarios. The architecture of the entire hybrid pipelined CPU-FPGA system is illustrated in Figure 1, indicating the functional division between the CPU and FPGA. The CPU is primarily responsible for text data acquisition, cleaning, and feature extraction (such as TF-IDF vectorization), while the FPGA performs parallel inference using the fixed-point quantized random forest model to enable real-time or near-real-time classification decisions.

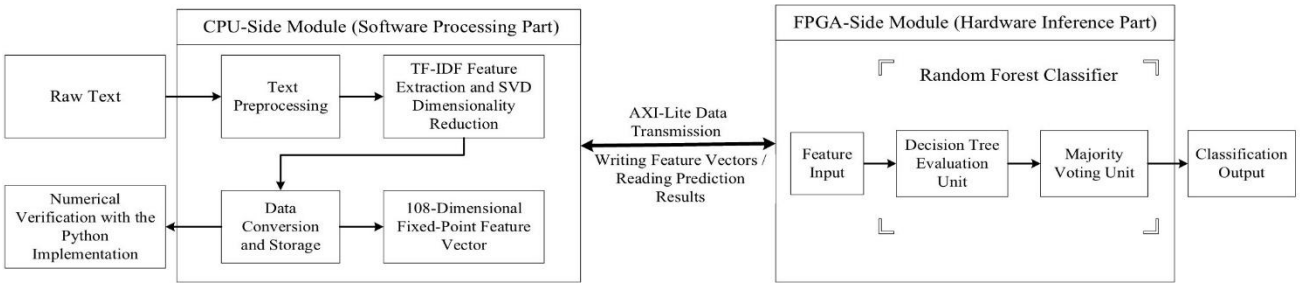

Fig 1: Architecture Diagram of Hybrid Pipelined CPU-FPGA System.

2.1 Data Collection and Preprocessing

For the complete implementation of text feature extraction and random forest as described in article [10], the data collection process during model training primarily relies on the ADReSS 2020 Challenge dataset [11]. This dataset, provided by an open competition, is specifically designed for Alzheimer's disease (AD) diagnosis and contains speech data from 54 Alzheimer's patients and 54 healthy controls, along with their corresponding complete transcription texts.

In subsequent collection of individual speech data, the adopted preprocessing method involves extracting complete speech transcription texts from each subject's audio recordings, followed by data cleaning procedures including the removal of noise symbols and ineffective stop words. For ease of subsequent analysis, all textual data were organized into a two-dimensional data structure, and a binary classification label was assigned to each sample to distinguish between patient and control groups. In this experimental setup, the dataset was partitioned using K-fold cross-validation, after which the processed textual data were fed into machine learning models for training and evaluation. The GridSearchCV method was employed for systematic hyperparameter tuning of the models, including parameters of the TfidfVectorizer such as maximum features, stop words configuration, term frequency limits, as well as parameters of the Random Forest classifier such as the number of decision trees, maximum tree depth, and the minimum samples required for node splitting. The algorithm utilized in the experiment included the TfidfVectorizer feature extractor for transforming the transcribed texts into numerical vector representations, and a Random Forest algorithm for handling the binary classification task. The optimized Random Forest model achieved a classification accuracy of 85.2% on this binary classification task, demonstrating the effectiveness of the combination of these two algorithms in Alzheimer's disease detection.

2.2 Lightweight Optimization and Hardware Adaptation of the Algorithm

In the original Python training pipeline, the model utilizes grid search (e.g., for parameters such as max_features, stop_words, and analyzer) to automatically adjust configurations, which often leads to the generation of large vocabularies, construction of complex random forest structures, and high-dimensional feature representations. To adapt to resource-constrained environments such as embedded systems and FPGAs, the authors first refined the vocabulary during the text feature extraction stage, in the TfidfVectorizer, they fixed the 'max_features' parameter at 934 while employing fixed configurations of stop_words='english' and ‘analyzer='word', this approach controlled the vocabulary size, reducing the complexity of subsequent feature computation and model deployment. Furthermore, to avoid the additional overhead associated with dynamic parameter tuning, the authors eliminated the grid search process and instead adopted fixed hyperparameter settings. Second, to further reduce feature dimensionality, the original 934-dimensional TF-IDF feature space was compressed to 108 latent semantic dimensions by applying truncated singular-value decomposition. This step not only preserved the essential information of the textual data but also significantly reduced the input dimensionality of the random forest classifier, thereby lowering the hardware resource demands for LUTs, BRAMs, and other FPGA components.

On this basis, the authors quantized the TF-IDF outputs into fixed-point form using a bespoke transformer, and substituted all floating-point operations with ap_fixed<32,16>. Fixed-point quantization not only significantly reduces computational complexity while maintaining model accuracy but also serves as a prerequisite for FPGA implementation. Moreover, to simplify the hardware implementation of the random forest model, the authors adopted a fixed configuration of 'RandomForestClassifier(n_estimators=10, max_depth=10, min_samples_split=5, min_samples_leaf=2, bootstrap=True) ', ensuring that both the number and depth of trees were strictly

constrained. Finally, the authors exported the lightweight core model parameters separately, including the refined vocabulary (stored as 'vocabulary.json'), IDF values (stored as 'idf_values.npy'), and the SVD matrix (stored as 'svd_matrix.npy'), this ensures that during deployment, these preprocessed parameters can be directly loaded, avoiding unnecessary computational overhead from repeated training.

In summary, by transforming large-scale search and high-dimensional input into a combination of fixed configurations, fixed-point quantization, and dimensionality reduction, the authors successfully reduced model complexity and memory consumption, thereby providing foundational support for FPGA deployment.

2.3 Fixed-Point Implementation of Text Feature Engineering (TF-IDF + SVD) in C++.

Traditional TF-IDF feature extraction is used in natural language processing (NLP) workflows to capture key information in textual data. To accommodate hardware acceleration requirements, the authors employs std::string for tokenizing and preprocessing input text. During tokenization, std::istringstream is used to split the text by whitespace, and std::transform is applied to convert all characters to lowercase to facilitate case-insensitive matching. A predefined stopword set (std::unordered_set<std::string>), including high-frequency words with no substantial informational value (e.g., “the” and “of”), is loaded in advance to enable fast lookup. Each word is checked against the stopword set, and if found, it is skipped. Although this data structure is difficult to synthesize directly on FPGA, it ensures filtering efficiency while maintaining implementation simplicity and scalability. The efficient lookup performance on the CPU side supports the overall preprocessing phase of the system.

During the term frequency (TF) computation phase, a fixed-length array is used to store the occurrence count of each word within the document. Each token in the text is processed by first undergoing character cleansing (removing non-alphanumeric characters) and then mapping to a predefined vocabulary (which assigns an index to each word), the corresponding index is used to increment the count in the array. In the TF-IDF computation step, the term frequency (TF) is obtained directly from word occurrence counts, which are then multiplied by precomputed inverse document frequency (IDF) values stored in an idf_table (an array of IDF values saved from the Python preprocessing stage), this process directly yields the TF-IDF values. In the first iteration, all word TF-IDF values are computed, and their squared sum is accumulated to provide the denominator for L2 normalization. In the second iteration, each normalized TF-IDF value is multiplied by the corresponding weight in a preloaded SVD matrix (svd_matrix [j][i]). The sum of these weighted values across all words constitutes the j-th dimension of the output vector. The final output vector, consisting of 108 dimensions, represents the dimensionality-reduced feature representation.

L2 normalization is implemented as follows: first, the squared sum of all nonzero TF-IDF values is computed, and its square root is taken to obtain the L2 norm. During the second pass, each TF-IDF value is divided by the L2 norm, yielding normalized values, which are then multiplied by a fixed-point scaling factor (128.0) and converted into the fixed_t type. L2 normalization ensures that the magnitude of the TF-IDF vector is standardized, preventing high-frequency words within a single document from disproportionately affecting subsequent calculations. The current implementation is entirely executed on the CPU using C++ code. By leveraging the standard library and precomputed vocabulary, IDF values, and SVD parameters, the entire process from text to feature vector computation is achieved efficiently.

The feature vector output after SVD dimensionality reduction is a 108-dimensional array, which serves as an input to hardware modules. This array can be transferred to FPGA computation units via an AXI-Lite interface or passed using simple memory-mapped access or direct memory access (DMA), depending on the system architecture and performance requirements. The module design employs fixed-point arithmetic, laying the groundwork for future FPGA-based hardware acceleration. When porting the algorithm to FPGA, parallel computing advantages can be leveraged to accelerate term frequency computation, TF-IDF calculations, and matrix multiplications, thereby further enhancing processing speed.

2.4 Hardware Acceleration for Random Forest

The random forest module primarily utilizes multiple decision trees to predict input 108-dimensional fixed-point feature vectors, ultimately determining whether a sample is classified as AD (Alzheimer’s disease) using a voting mechanism. The logical process is as follows: the function 'evaluate_tree' takes a decision tree node array (nodes) and an input feature vector (features) as inputs, starting from the root node (node_id = 0). Using a fixed-depth loop (e.g., with a maximum depth of 32), at each level, the algorithm compares feature values to decide whether to traverse the left or right subtree. Upon reaching a leaf node, the node stores a parameter 'value_diff', which determines the tree’s prediction result by checking whether it is greater than zero (returning a Boolean value: true for AD, otherwise for non-AD). For non-leaf nodes, the algorithm compares the feature value at the specified index with the node threshold (node.threshold) to determine the index of the next node. This part directly employs fixed-point number comparison, eliminating the need for additional conversions, thus ensuring both efficiency and precision.

The traversal loop employs the #pragma HLS PIPELINE directive, ensuring that each comparison operation is executed in a pipelined manner, thereby reducing latency. Additionally, the #pragma HLS INLINE off directive controls the function inlining strategy, ensuring that the implementation of 'evaluate_tree' aligns with hardware optimization requirements and avoids excessive inlining, which could otherwise consume additional resources.

Within the 'predict' function, the evaluations from 10 decision trees are executed by invoking evaluate_tree. The #pragma HLS UNROLL directive instructs the compiler to unroll these 10 calls into a parallel hardware implementation. Each tree returns a Boolean value (converted into an integer and accumulated), and the final prediction is determined through a majority voting mechanism: if the vote count is greater than or equal to 5, the result is classified as AD; otherwise, it is classified as non-AD.

To accommodate the hardware platform, the prediction function applies the #pragma HLS INTERFACE s_axilite directive, indicating that both the input array features and the prediction result are transmitted via the AXI-Lite interface. The AXI-Lite interface is well-suited for the transmission of small-scale configuration data and control signals, as it is simple to use and requires minimal resources. Furthermore, by employing the #pragma HLS ARRAY_PARTITION directive,

the features array is partitioned according to a loop factor, thereby enhancing parallel access capability.

3. HARDWARE SIMULATION AND SYNTHESIS VERIFICATION

The correctness of the algorithm serves as a prerequisite for effective hardware acceleration, it is essential to validate, at the software level, that the feature extraction and classification logic currently implemented is correct. Specifically, the entire pipeline – from raw text input through TF-IDF feature extraction to prediction – must be verified to ensure accurate hardware acceleration on FPGA devices.

The authors executed the script (get_stop_words.py) to extract the standard ENGLISH_STOP_WORDS set provided by scikit-learn and subsequently saved these words into a local text file named stopwords_en.txt. This step ensures that common irrelevant words are automatically filtered out in subsequent word frequency analyses, allowing the TF-IDF feature extraction process to focus more precisely on the key information.

A lightweight Random Forest model (RandomForestClassifier) was trained using the previously implemented script (train_light_model.py). The training process involved learning the IDF parameters and SVD matrix through a pipeline comprising TfidfVectorizer, TruncatedSVD, and a custom FunctionTransformer (named to_fixed_point) to enable fixed-point representation. Upon completion of training, the resulting core files included the pipeline model (light_model.joblib), IDF parameters (idf_values.npy), the SVD principal component matrix (svd_matrix.npy), and the vocabulary mapping table (vocabulary.json) associated with the TfidfVectorizer. Subsequently, the authors executed convert_params.py to convert critical parameters from the trained model into a header file format (model_params.h) suitable for inclusion in C++, thereby completing the parameter export.

The file 'main.cpp' was implemented to test the complete end-to-end functionality, demonstrating the entire process from raw text input to final prediction. The process first involves obtaining a 108-dimensional feature vector via the 'tfidf_transform()' function, followed by performing Random Forest inference using the 'predict()' function. The test input was defined as a descriptive passage containing typical syntactic errors characteristic of Alzheimer's patients. To facilitate comparison with Python-generated outputs, two functions were implemented: 'save_features()', which stores the 108-dimensional fixed-point feature vector in binary format in the file "cpp_features.bin", and 'save_features_txt()', which saves the feature vector in textual format to "cpp_features.txt" for subsequent numerical analysis. An executable file was generated using the command:

g++ -std=c++11 -I. -I"E:/vitis/Vitis_HLS/2022.2/include" main.cpp random_forest.cpp tfidf.cpp -o end2end.exe

Upon execution, the terminal displayed "End-to-end prediction results in C++: 1", and produced the files 'cpp_features.bin' and 'cpp_features.txt' for subsequent comparison with Python-based experimental results. A classification outcome of "1" indicates that the test text was correctly identified as belonging to Alzheimer's patients.

To facilitate verification, the authors designed and implemented a Python-based inference module (predict_light_model.py) to replicate the inference pipeline constructed during training, performing model inference on identical input texts. Within this module, the function 'preprocess_text()' reproduces the text cleaning logic used in training. Regular expressions (regex) were employed to remove time stamps (e.g., patterns like '\x15\d_\d\x15'), annotations (content within brackets), as well as special characters such as tabs, line breaks, and angle brackets, thereby standardizing the input text and ensuring consistency and robustness in subsequent feature extraction steps. To allow for comparison with the feature vectors generated on the C++ side, the resulting 108-dimensional feature vector was exported to a text file named 'test_features.txt'. In the main function, the same input text as used in the C++ test case was selected, and the prediction function was invoked to obtain both the prediction result and the corresponding feature vector. Finally, the program printed the prediction result, displaying "The Python prediction result: 1," along with the dimensional information of the feature vector, and saved the generated feature vector file.

To verify data consistency between the C++ and Python implementations, the feature vector files generated from both sides ("cpp_features.txt" and "test_features.txt") were loaded using NumPy's 'np.loadtxt()' function. The loaded vectors were then compared using the 'cosine_similarity' function provided by the scikit-learn library. Cosine similarity, a commonly used measure for assessing vector similarity, quantifies the similarity between vectors based on their direction, with values ranging from -1 to 1, where values closer to 1 indicate greater similarity. The obtained similarity score was 0.9990 (target threshold ≥ 0.99), demonstrating a high level of directional consistency between the vectors produced by the C++ and Python implementations. This result confirms that the TF-IDF and SVD computation logic has been correctly implemented, validating numerical consistency and the correctness of the end-to-end processing pipeline.

3.1 Hardware preparation and Vitis compilation

Run the code to convert 'vocabulary.json' into a C++ header file 'vocabulary.h', thus statically embedding the vocabulary mapping table; export the random forest tree structure from 'light_model.joblib' into 'trees.json', and subsequently generate 'tree_params.h'; convert 'idf_values.npy' and 'svd_matrix.npy' into 'idf_values.h' and 'svd_matrix.h' respectively, facilitating their referencing as read-only tables within HLS.

Import 'testbench.cpp', 'random_forest.cpp', 'tfidf.cpp', along with the previously generated header files (such as 'tree_params.h', 'vocabulary.h', etc.) into Vitis for High-Level Synthesis (HLS). After running the C Simulation (CSIM), inference results identical to the software-side were obtained: FPGA predict result: 1; no errors were returned, confirming functional correctness.

When Vitis HLS performs C Simulation and C Synthesis on C++ (.cpp) files, it generates corresponding reports to evaluate functional correctness, resource utilization, latency, and performance metrics. C Simulation verifies the functional correctness of the C/C++ code, ensuring that the algorithm logic is error-free. C Synthesis, on the other hand, converts the C/C++ code into an RTL-level (Verilog/VHDL) hardware description and assesses the resource consumption, timing, and latency of the hardware implementation. The Synthesis Report is shown in Figure 2.

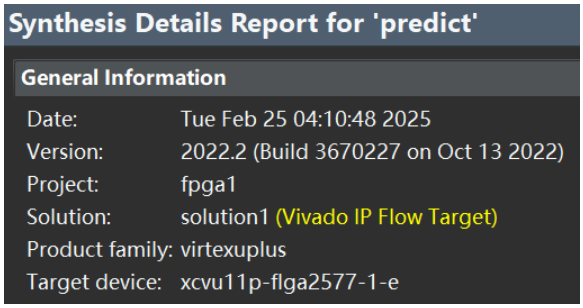

Fig 2: Vitis HLS C Synthesis Report.

3.2 Experimental Results and Analysis

Table 1 presents the synthesis timing results of the FPGA design in the Vitis HLS environment. As shown in Table 1, the target FPGA device used in this experiment is xcvu11p-flga2577-1-e, with a target clock period of 10 ns. The actual estimated clock period after synthesis is 6.368 ns, with an uncertainty of ±2.70 ns. The synthesis timing analysis shows that the total latency of the random forest inference function predict() is approximately 11192~11193 clock cycles. At the target clock frequency of 100 MHz, this implies that a single inference task takes about 112 µs, whereas based on the actual estimated clock period, the latency reduces to 71.3 µs. From these data, the derived FPGA operating frequency is calculated to be around 157 MHz (Derived clock frequency), which is obtained by taking the reciprocal of the clock period (1 / clock period).

Table 1. Post-synthesis Timing Summary

Parameter	Value	Unit
Target Device	xcvu11p-flga2577-1-e	—
Target Clock Period	10.0	ns
Estimated clock period	6.368±0.270	ns
Derived clock frequency	157±6.7	MHz
Latency (cycles)	11192~ 11193	cycles
Latency @ 100 MHz	112	µs
Latency @ estimated T	71.3	µs

Next, FPGA resource utilization can be observed based on Table 2 and the stack figure 3. The FPGA design uses a total of 370,421 LUTs (28%), 471,034 FFs (18%), and 546 DSP blocks (5.9%), while BRAM and UltraRAM are not utilized. As can be seen from the resource distribution figure 3, the primary resource consumption in the design is concentrated on LUTs and FFs, FPGA-internal Block RAM (BRAM) resources have not been utilized to store data. Overall, only 5% to 30% of resources are occupied on the ultra-large FPGA device. The decision tree structure (Random Forest) is mainly stored in LUTRAM, while the core computation utilizes DSP and LUT resources. The LUTRAM stores various tree node information, comprising a total of 30 ROM blocks. Most critical logic (including vote counting and feature comparison) uses fixed-point arithmetic, requiring a certain number of DSP and LUT resources allocated by HLS for addition and multiplication operations. In the future, it would be possible to optimize the computational logic to map more computing tasks onto DSP resources.

Table 2. Post-Synthesis Resource Utilization

Parameter	Value	Device Capacity	Utilization (%)
LUT	370,421	1,296,000	28%
FF	471,034	2,592,000	18%
DSP Blocks	546	9216	5.9 %
Block RAM (18Kb)	0	4032	0%
UltraRAM	0	960	0%

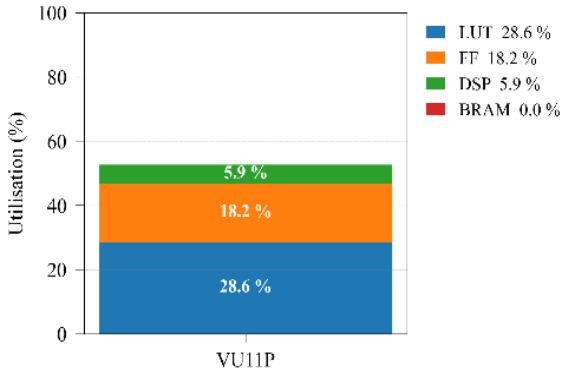

Fig 3: Post-synthesis resource-utilization breakdown.

Further, the authors compared the single-inference performance between CPU and FPGA implementations. As shown in Table 3, the total elapsed time parameter generated by the C Simulation indicated the total duration from the start to the end of the simulation process was 30.505 seconds. Meanwhile, the FPGA execution time was calculated by multiplying the clock period (Clock Period parameter from the csynth.rpt report) by the latency (number of cycles), resulting in approximately 0.0000713 seconds. Consequently, the authors calculated an idealized speedup of approximately 427,840×. However, this speedup is highly idealized since it compares the "simulation environment plus pure hardware computation." In practical deployments, pure inference time on the CPU is typically much less than 30 seconds, and the FPGA requires data transfers through DDR, PCIe, or AXI interfaces; hence, transfer latency must be considered in real-world scenarios.

To achieve more realistic results, the authors compiled and ran the same inference code using g++ with -O2 optimization on a 2.9 GHz CPU, obtaining an average single inference latency of 4.713 µs. Compared with the FPGA synthesis report, to ensure stable design operation, the synthesis tools typically use a target clock of 10 ns, corresponding to a frequency of 100 MHz. Under this frequency, the FPGA hardware latency per inference was 112 microseconds (µs). In this case, the CPU was approximately 24 times faster than the FPGA, primarily because the FPGA clock frequency was relatively low and had not yet leveraged batch parallelism or other methods to exploit hardware potential fully. When calculating the real FPGA latency, using 71.3 µs represented using the "Estimated Clock Period" from the HLS report, optimistically assuming the final implementation could also achieve approximately 157 MHz and disregarding board-level clock convergence and routing margins. Conversely, using 112 µs was based on the constrained clock period of 10 ns (100 MHz), which is more conservative and typically serves as the realizable target provided by most design flows before place-and-route.

Table 3. Inference Time and Speedup Comparison

Comparison basis	CPU Time	FPGA Time	Speedup Calculation	Conclusion
Real inference latency (optimistic)	4.713 μs	71.3 μs	4.713 ÷71.3 ≈ 0.066	FPGA is ≈ 15× slower
Real inference latency (conservative)	4.713 μs	112 μs	4.713 ÷112 ≈ 0.042	FPGA is ≈ 24× slower
Tool-level "idealized" comparison	30.505 s	71.3 μs	30.505 s ÷ 71.3 μs ≈ 427840	Reflects only tool-level simulation overhead, not actual hardware performance

Overall, the current FPGA design exhibits favorable performance concerning functional correctness, resource utilization efficiency, and timing compliance. The top-level interface adheres to the AXI4-Lite standard, with features as input and result as output, interacting via register mapping. Hardware resource utilization complies with FPGA resource constraints; LUT, FF, and DSP usage do not exceed the target device's capacity. These experimental results indicate that the advantage of FPGA lies not in single-inference latency but rather in throughput, energy efficiency, and real-time processing capabilities. The effectiveness of FPGA acceleration depends on multiple factors, including task size, degree of data parallelization, hardware resources, and I/O bandwidth.

Through the aforementioned multi-stage experiments and toolchain integration, the authors successfully completed the synthesis verification of the random forest inference IP in Vitis HLS and performed numerical and functional consistency comparisons with the Python/CPU implementations. The current validation process has achieved a high level of maturity in terms of functional correctness and data consistency.

4. CONCLUSION

This study designed and demonstrated a CPU–FPGA-based proof-of-concept framework to accelerate random-forest-based text classification tasks. The framework employs a lightweight random forest model and incorporates strategies such as algorithm lightweighting, fixed-parameter configuration, fixed-point quantization, and dimensionality reduction, thereby realizing algorithm deployment on FPGA. Experimental results show that the framework achieves high functional consistency with the Python reference implementation, with a cosine similarity of 0.9990, while occupying only approximately 28% of LUTs and 5.9% of DSP resources on a Xilinx VU11P device. This heterogeneous architecture offloads the inference kernel to hardware implementation, achieving a post-synthesis latency of approximately 112 μs at a conservative target frequency of 100 MHz, while this latency can be further reduced to 71.3 μs based on the estimated clock period of 6.368 ns (≈157 MHz) from the synthesis report.

However, the current prototype still executes TF-IDF and SVD feature extraction on the host CPU and has not undergone place-and-route or power analysis. Therefore, system-level latency and energy efficiency remain suboptimal. Future research will focus on increasing parallelism on the FPGA, using HLS directives to parallelize computations across each decision tree and each feature dimension, performing power evaluation after place-and-route, utilizing BRAM/URAM to cache model parameters and intermediate data to fully exploit FPGA resources, and further attempting to migrate the TF-IDF and SVD preprocessing to FPGA to eliminate PCIe/AXI transfer bottlenecks. It is anticipated that these improvements will further enhance throughput and strengthen the system's end-to-end hardware-accelerated performance.

5. REFERENCES

[1] Andrzej Janowski. Natural Language Processing Techniques for Clinical Text Analysis in Healthcare. Journal of Advanced Analytics in Healthcare Management, 7(1):51–76, Mar. 2023.

[2] Rehana H, Çam NB, Basmaci M, Zheng J, Jemiyo C, He Y, Özgür A, Hur J. Evaluation of GPT and BERT-based models on identifying proteinprotein interactions in biomedical text. ArXiv, 2023: arXiv: 2303.17728 v2.

[3] Martin Wisniewski, Lucas, Jean-Michel Bec, Guillaume Boguszewski, and Abdoulaye Gamatié. Hardware Solutions for Low-Power Smart Edge Computing. Journal of Low Power Electronics and Applications, 12(4):61, 2022.

[4] Movva, Rajiv, Jinhao Lei, Shayne Longpre, Ajay Gupta, and Chris DuBois. Combining Compressions for Multiplicative Size Scaling on Natural Language Tasks. arXiv preprint, 2022. arXiv:2208.09684.

[5] Liao, Youqi, Shuhao Kang, Jianping Li, Yang Liu, Yun Liu, Zhen Dong, Bisheng Yang, and Xieyuanli Chen. Mobile-seed: Joint semantic segmentation and boundary detection for mobile robots. IEEE Robotics and Automation Letters, 2024.

[6] Liu, Linyuan, Haibin Zhu, Tianxing Wang, and Mingwei Tang. A Fast and Efficient Task Offloading Approach in Edge-Cloud Collaboration Environment. Electronics, 13(2): 313, 2024.

[7] Zhang, Chaoyu, Hexuan Yu, Yuchen Zhou, and Hai Jiang. High-Performance and Energy-Efficient FPGA-GPU-CPU Heterogeneous System Implementation. In Advances in Parallel & Distributed Processing, and Applications: Proceedings from PDPTA'20, CSC'20, MSV'20, and GCC'20, pages 477–492. Springer, 2021.

[8] Mouri Zadeh Khaki A, Choi A. Optimizing Deep Learning Acceleration on FPGA for Real-Time and Resource-Efficient Image Classification. Applied Sciences, 15(1), 2025.

[9] Hamza Khan, Asma Khan, Zainab Khan, Lun Bin Huang, Kun Wang, and Lei He. NPE: An FPGA-based Overlay Processor for Natural Language Processing. arXiv preprint arXiv:2104.06535, 2021.

[10] Vishniakou U.A and Chuyue Yu. Using Machine Learning for Recognition of Alzheimer's Disease Based on Transcription Information. Reports of BSUIR, 21(6): 106–112, 2023.

[11] Saturnino Luz, Fasih Haider, Sofia de la Fuente, Davida Fromm, and Brian MacWhinney. Alzheimer's dementia recognition through spontaneous speech: The ADReSS challenge. arXiv preprint arXiv:2004.06833, 2020.