Scalable Hardware Architecture for fast Gradient Boosted Tree Training

TAMON SADASUE^{1,2,a)} TAKUYA TANAKA¹ RYOSUKE KASAHARA¹ ARIEF DARMAWAN²
TSUYOSHI ISSHIKI^{2,b)}

Received: June 3, 2020, Revised: September 8, 2020,
Accepted: October 23, 2020

Abstract: Gradient Boosted Tree is a powerful machine learning method that supports both classification and regression, and is widely used in fields requiring high-precision prediction, particularly for various types of tabular data sets. Owing to the recent increase in data size, the number of attributes, and the demand for frequent model updates, a fast and efficient training is required. FPGA is suitable for acceleration with power efficiency because it can realize a domain specific hardware architecture; however it is necessary to flexibly support many hyper-parameters to adapt to various dataset sizes, dataset properties, and system limitations such as memory capacity and logic capacity. We introduce a fully pipelined hardware implementation of Gradient Boosted Tree training and a design framework that enables a versatile hardware system description with high performance and flexibility to realize highly parameterized machine learning models. Experimental results show that our FPGA implementation achieves a 11- to 33-times faster performance and more than 300-times higher power efficiency than a state-of-the-art GPU accelerated software implementation.

Keywords: FPGA, acceleration, machine learning, Gradient Boosted Tree, hardware description language

1. Introduction

Although Deep Neural Networks have proven their powerful learning ability and expandability in machine learning, computer vision and natural language processing, gradient boosted tree (GBT) algorithm is still the most effective and standard algorithm for tabular datasets which includes a wide variety of feature values, different filling rates, and often missing data along with its ability to analyze the attribution of features [1]. Therefore, most learning models used machine learning competitions use a GBT as a base algorithms or multiple ensembled learning models [2], [3].

In the Web-search ranking [4], [5], dataset is composed of the user's query keywords, cookie data, other tagged information, the search history, and earlier web-search results have already reached one millions samples and several hundred features, and continuing to increase year by year. In industry, a huge number of data samples need to be handled. In the FinTech fields such as stock movement and online bidding, it is necessary to frequently update a model that reflects the latest trends. In such a situation, a high-speed training method is required for the purpose of trying as many learning models and hyperparameter searches as possible.

In this paper, we propose a fully pipelined, efficient, high-speed

logic architecture to speed up the GBT training, by which the entire learning process can be realized on an FPGA. The design of a high throughput pipelined logic that can adapt to changes in data size, a variety of data parallelisms, and a variety of loss functions in the search for the optimal hyperparameters for a GBT is quite challenging, and very few studies have reported on GBT training acceleration on FPGAs. Our high throughput pipelined architecture with flexible configuration parameters is designed on our C2RTL design framework, in which the GBT algorithm is fully described in C++ and the system-level RTL descriptions are automatically generated directly from the C++ descriptions using C2RTL.

This paper is organized as follows. In Section 2, related work on a GBT training algorithm implementation are described in which we also clarify the key contributions of this paper. In Section 3, the basic GBT algorithm is overviewed, followed by a description of the proposed hardware architecture in Section 4. In Section 5, the design methodology of our C2RTL framework is described. Section 6 describes the implementation results and comparisons to a GPU implementation. Finally, in Section 7, the proposed architecture and its results are summarized.

2. Related Work

Regarding previous work on GBT training acceleration techniques, multiple CPU threads, distributed system and GPU accelerated method [6], [7], [8], [9], [10] have been reported, and some of their implementations are being widely used. However, the degree of speedup remains small compared to that of Random Forest, which is another popular decision tree ensemble technique.

¹ Innovation R&D Division, RICOH Company, Ebina, Kanagawa 243-0460, Japan

² Information and Communications Engineering, Tokyo Institute of Technology, Ohta, Tokyo 152-8550, Japan

^{a)} tamon0215@sky.email.ne.jp

^{b)} issiki@ict.e.titech.ac.jp

The GBT algorithm produces a new tree to compensate for past losses, resulting in high accuracy and fast convergence. However, unlike Random Forest, where all trees can be independently constructed resulting in a high degree of parallelism, it is difficult to exploit parallelism at the tree level owing to dependency between consecutive tree constructions. In addition, GBT training process needs a high computation cost for an exhaustive split condition search, and the memory bandwidth can become a large bottleneck for such a process. Therefore, GPU acceleration for GBT training has limited success compared to Random Forest training.

An energy efficient GBT training method on an FPGA was first reported [11] as a prior aspect of this study. This prior approach had several drawbacks such as requiring all sample feature data in the expensive on-chip RAMs, and allowing only a fixed set of algorithm parameters in the FPGA implementation. These drawbacks were mostly due to the hardware design methodology using HDLs, which requires significant effort in the overall system design as well as in exploring different system-level architectural options.

The main contribution of this paper is the significant extensions of our prior approach, that is, a highly parameterized hardware implementation that allows an efficient operation even on low-cost FPGA devices with a small memory capacity. This was achieved by first describing the entire GBT training system in C++ along with all configuration parameters such as the number of samples, the feature dimensions, the maximum tree depth, the loss functions gain evaluation methods and the memory interface between external DRAM and on-chip RAMs. These C++ descriptions of the entire GBT training system are then directly synthesized into a system-level RTL using our C2RTL design framework, resulting in a highly efficient FPGA implementation in terms of both computational throughput, power consumption, and hardware flexibility.

3. Gradient Boosted Tree Algorithm

3.1 Overview of GBT Training Algorithm

The GBT training algorithm for consecutively constructing new trees is shown in Algorithm 1. The first process is to search for the best split condition evaluated by a gain formula that represents the separation degree of the classification result (lines 6–13), the second is to split the samples contained in the node into two child nodes according to the condition found (line 18), and after the above two processes are repeated until reaching the maximum depth of the tree, the predicted values of the samples are updated with the weight of the leaf node calculated using the teacher labels included in that leaf node (lines 20–22). Then, for the next tree, the gain is calculated based on the difference between the updated predicted value and the training value, the prediction error will be compensated at each additional tree construction.

The gain for the split condition candidates at the current node is evaluated using Eq. (1) where γ is the regularization term. In addition, G_L and G_R the sum of gradient g_i of all current samples in the left node and right node, respectively. Likewise, H_L and H_R are the sum of Hessian h_i . Gradient g_i is the first-order derivative of the loss function denoted by $l(y, (y_i))$ and Hessian h_i is the second-order derivative of the loss function, as shown in

Algorithm 1 Making a new tree of GBT training

Input: training samples
Input: F: list of features
Input: maxdepth: maximum depth
Output: t: decision/regression tree

```

1: INITTREE(t)
2:  $N \leftarrow \text{RootNode}(t)$ 
3: for all  $n \in N$  do
4:    $S_n \leftarrow \text{SamplesInNode}(n)$ 
5:    $\text{maxgain} \leftarrow 0, \text{threshp} \leftarrow (0, 0)$ 
6:   for all  $f \in F$  do
7:     for all  $v \in \text{FeatureValues}(f)$  do
8:        $\text{gain} \leftarrow \text{COMPUTEGAIN}(S_n, f, v)$ 
9:       if  $\text{gain} > \text{maxgain}$  then
10:         $\text{maxgain} \leftarrow \text{gain}, \text{threshp} \leftarrow (f, v)$ 
11:       end if
12:     end for
13:   end for
14:    $t \leftarrow t \cup (n, f, v, \text{maxgain})$ 
15:   if  $\text{maxgain} < 0$  then
16:      $\text{UPDATELEAFNODE}(n, N)$ 
17:   else
18:      $(n1, n2) = \text{SPLITNODE}(n, N, f^*, v^*)$ 
19:   end if
20:   if  $\text{depth}(n) + 1 = \text{maxdepth}$  then
21:      $\text{UPDATELEAFNODE}(n1), \text{UPDATELEAFNODE}(n2)$ 
22:   end if
23: end for
    
```

Eq. (2).

$$\text{Gain} = \frac{1}{2} \left[\frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{(G_L + G_R)^2}{H_L + H_R + \lambda} \right] - \gamma \quad (1)$$

$$g_i = \frac{\delta l(y, \hat{y})}{\delta \hat{y}}, \quad h_i = \frac{\delta^2 l(y, \hat{y})}{\delta \hat{y}^2} \quad (2)$$

When samples reach a certain leaf node j , the prediction values of these samples will be updated by adding a leaf weight w_j^* given by Eq. (3), where $G_j = \sum_{i_j \in I_j} g_i$, $H_j = \sum_{i_j \in I_j} h_i$, and I_j is the set of samples reached at leaf node j . Then these new prediction values will be used for constructing the next tree.

$$w_j^* = \frac{-G_j}{H_j + \lambda} \quad (3)$$

3.2 GBT Training Hotspots

The most compute-intensive part of the algorithm is finding the split condition at each node. This part can be further divided into two parts: (A) sorting all samples at every feature, and (B) a maximum gain search over all threshold candidates. Since the processing time of the GBT algorithm is dominated by these two hotspots, accelerating these parts in hardware results in significant throughput improvement.

3.2.1 Sorting based versus Histogram based Split Finding Algorithm

As reported in previous work [6], [12], sorting all feature values in advance in order to find the optimal split condition from all possible candidates requires heavy computing resources when the number of distinct feature values is large, but a histogram based algorithm can reduce this computing cost drastically. Therefore, we quantize the feature values to fit them into a fixed number of

bins for each attribute and make a histogram by accumulating the gradient and hessian for each bin, finally choosing the threshold that gives the maximum gain.

3.2.2 Acceleration of Gain Calculation

The gain calculation in Eq. (1) is needed for all features and threshold candidates during the split condition search, resulting in a large computational load. On multi-core CPUs and GPUs, acceleration of the gain calculation can be achieved by data parallelisms applied on different features and thresholds, although this approach may create a data transfer bottleneck of distributing the sample data over multiple processing elements. Instead, we directly connect a dedicated pipelined gain calculation hardware to the gradient/hessian histogram memories to avoid any data transfer bottleneck and achieve a sufficiently high gain calculation throughput.

4. Hardware Architecture

Our approach has five main characteristics. The first is the histogram based approach mentioned earlier. A histogram-based approach is well suited to hardware implementation because not only does it reduce computing cost of a comparison from $O(\#sample * \#feature)$ to $O(\#bin * \#feature)$, the memory consumption also becomes a constant ($\#bin * \#feature$) and thus it can be easily stored in on-chip memory. The second characteristic is a pipeline architecture, which avoids a random memory access and often results in a degraded performance to a memory bandwidth bottleneck, and achieves a high throughput in most parts of the training process. The third characteristic is a pointer memory architecture that keeps both a high throughput and low memory consumption at the same time. The fourth characteristic is the efficient parallelism based on the throughput per hardware area. The last is a methodology that uses our new efficient design tool ‘C2RTL’, enabling flexibility and scalability to support various types of hyperparameters and environmental constraints (more details of this are provided in the next section).

4.1 Pipeline Architecture

Our architecture consists of the modules shown in the Fig. 1. There are four modules holding on-chip memory. The pointer memory module holds the pointers for the samples, the prediction memory module holds the labels and the predicted values for the samples, and the histogram module stores the accumulated gradient and hessian values for each bin in on-chip memories. The feature memory holds the number of feature values determined by the number of fetchable feature dimensions and the samples. The three main processes that are repeatedly executed are the split finding process, splitting process, and update process, that correspond to the colored parts in the Fig. 1. The split finding process and the splitting process are repeated as many times as there are nodes at a certain depth. This set of processes are further repeated as many times as the maximum depth.

These three processes must be sequential, because the splitting process cannot be started until the split condition search after histogram generation is completed, and the predicted value cannot be updated until the samples entering each node are determined. However, as shown in Fig. 2, the sub-processes within

Fig. 1 Overview of system module architecture.

Split Finding Process

Splitting Process

Updating Process

Fig. 2 Pipeline of three processes.

each process are pipelined to handle one sample per clock cycle (the number in the sub-process block indicates the processing sample). This architecture provides the most effective acceleration while avoiding duplicate and random access to the memory. Here, the gradient calculation module has delay of 4 cycles because this module is further pipelined with for stages internally to increase the clock frequency in this case.

4.1.1 Split Finding Process

Feature values for each sample are read from feature memory, the predicted value is read from prediction memory, and the gradient and hessian values are simultaneously calculated from the label and the predicted value through the CalcGrad module. These values are fed to the histogram module, which accumulates these gradient and hessian values at each histogram bin.

After gradient and hessian values corresponding to all samples are accumulated, we search for the condition that maximizes the gain after splitting using the selected feature and threshold. This gain is evaluated using Eq. (1), where the first two terms represent the separation degree of classification at the two child nodes, and the third term represents the separation degree of classification at the original node. The last term is the regularization term. Here, using the maximum gain module, we employ a full exhaustive search where Eq. (1) is evaluated for all threshold candidates located at the histogram bin boundaries. The maximum gain module spends constant cycles according to the number of bins and the feature dimensions.

4.1.2 Splitting Process

After the split condition is found, every samples in the node will be split into two nodes at the next depth. Node splitting

Fig. 3 Example of pointer memory read/write.

is performed by reading the corresponding feature values of all samples in the node from the feature memory (through the pointer memory mechanism described in Section 4.2), comparing the values against the splitting threshold, and updating the pointer memory to reflect whether each sample resides in the left or right child node. These processes are executed by pipeline one sample per cycle. After all nodes at current depth are processed, the split finding and splitting process continue to the next depth.

4.1.3 Updating Process

After reaching the maximum depth, in the prediction memory module, the predicted value of every sample is updated by adding the weight of the leaf node that contains the sample while reading the sample index and node index from the pointer memory. The weight of the leaf node is calculated using Eq. (3).

4.2 Pointer Memory Structure

Although many software tree structure implementations are possible such as dynamic 2D arrays or a linked list, a hardware tree structure implementation requires special care for sustaining high throughput processing. Here, we employ the pointer memory which is an array of sample indices used as address values for accessing the on-chip feature memory. The tree structure is maintained by a set of registers that indicate the boundaries of the tree nodes on the array. Two sets of such arrays are used in a ping-pong fashion to facilitate the tree splitting process.

At each splitting stage, the current sample indices in one RAM array are read one by one, where each sample index is then written to the other RAM array, either from the left side (left split) or from the right side (right split), as shown in Fig. 3 (1). After all sample indices have been relocated from one RAM to the other, the roles of the read/write RAMs are reversed at the next node splitting shown in Fig. 3 (2). This pointer memory scheme keeps the memory consumption at a constant size independent of the tree depth, and is effective in a hardware tree structure implementation.

4.3 Feature Prefetching

Because it is difficult to retain the feature data of 100,000 or more samples in on-chip RAM of a small FPGA, the features must be stored in external memory. In addition, to sustain the process throughput of one sample per cycle, feature fetch module

shown in Fig. 1 with a DMA engine transfers the required features from the external memory into the on-chip RAM. Input labels are also transferred beforehand into the on-chip RAM.

Depending on the on-chip RAM size, sample size, and feature size, different prefetching strategies are considered. The first case is the most simple in which all features of all samples can be stored in on-chip RAM. In this case, DMA completes a prefetching of all data at once before making the trees. The second case is when it is not possible to hold all features ($\#sample * \#feature$), but is possible to hold all the feature data for several attributes ($\#sample * K$). In this case, the feature memory can be used as a temporary buffer, and the prefetching is overlapped with the histogram generation and the maximum gain search processing. To execute the splitting stage immediately after the split finding stage is completed, another memory is prepared to hold the feature data giving the maximum gain. The third case is in which the on-chip memory cannot store ($\#sample$) the feature data, we can accumulate the gradient and hessian of all samples with feature values by fetching several times repeatedly. We assume all samples input labels, and prediction values can be held in on-chip RAM.

4.4 Parallelism

Parallelism is necessary to further accelerate each stage, and several types of parallelism can be considered here. Tree level parallelism is not applicable because a new tree depends on the construction of previous trees. To consider the node level parallelism with our architecture, one choice is to divide the pointer memories such that multiple nodes residing in different pointer memories can be processed in parallel. However, tree splitting may result in a highly unbalanced tree, which may introduce a large overhead in the total pointer memory size. Another way is to prepare multiple histogram generation modules and gain calculation modules for parallelizing these processes. This approach may also result in a large circuit overhead. Overall, we have observed little advantage of employing node level parallelism.

Based on the above observation, we adopted feature level parallelism and sample interleaving parallelism instead. feature level parallelism involves processing N feature dimensions in parallel using the same number of histogram generation modules and gain calculation modules, which can directly contribute to the speedup of the split finding stage. Sample interleaving parallelism uses the technique of dividing all samples in an interleaving fashion and also duplicating the pointer memory, feature memory, and prediction memory for simultaneously accessing the interleaved samples in parallel. We can apply the same sets of processes to these interleaved sample groups independently, and then merge these multiple set of gradient and hessian histograms before a gain calculation. Figure 4 shows an example of a two-sample interleaving case.

4.5 Support for Data Subset and Random Sampling

When we want to terminate the training early (early-stop) based on certain criteria, we use a data subset for calculating the validation error. Or when we want to increase the effect of the regularization and mitigate the risk of an over-fitting, we use ran-

Fig. 4 Example of duplicated modules for 2 sample interleaving.

dom sampling each time a new tree is made.

For both of these cases, we can pick the sample feature values randomly from sequential memory in external memory into different feature memory modules. This is done in a similar way as the interleaving method described before. For example, we make four sets of pointer memory and feature memory modules: three sets are used to hold the training subset, one set is used to hold the validation subset. A feature memory module for validation does not participate in a split finding pipeline. In a random sampling case, one of four or one of two samples will be chosen randomly to participate in the split finding stage. For this function, the index of the feature memory module that determines the destination memory is given by the pseudorandom generator within the feature fetch module.

4.6 Numerical Computation

Because floating-point arithmetic takes a high cost for both hardware and the gate delay, we replaced it with fixed-point arithmetic involving all variables and constants including the training value, prediction value, gradient, hessian, histogram, and leaf weight. We use 8 bits for the fraction part in most cases.

4.6.1 Loss Function

The most commonly used loss function for classification is logistic regression loss, and predicted value at that time is calculated by the sigmoid function denoted in Eq. (4). Because the sigmoid function includes the exponential term, we replaced the sigmoid function with an approximate expression by using a piecewise linear approximation [13] with the division of seven symmetrical sections and the clipping of infinite areas. In addition, we tested replacing the loss function, with the squared error function which is commonly used for regression problems.

$$S(x) = \frac{1}{1 + \exp(-x)} \quad (4)$$

4.6.2 Gain Calculation

The gain calculation is the most compute-intensive part. To avoid using a divider, we transform the variable part of the original equation in Eq. (1) into Eq. (5). We convert the divider into subtraction in the base 2 logarithmic space, and then convert back to the linear space through exponential arithmetic. The same transformation technique is applied to the left leaf weight W_L given by Eq. (6) and the right leaf weight W_R given by Eq. (7), these are originally Eq. (3).

Logarithmic arithmetic is implemented by counting the most significant bit (MSB) and exponential arithmetic, which is implemented through a shift operation (Fig. 5). The fraction part is ap-

Fig. 5 Transformation of gain calculation.

Table 1 Comparison of gain calculation implementation (pipelines = 4).

Implementation	LUT	FF	DSP	Clock Freq.
Naive (Divider)	4438	705	4	100 MHz
Our approximation	1601	379	0	250 MHz

proximated with a linear interpolation. Despite the low hardware area and little gate delay, a better resolution and approximation can be achieved compared to a naive linear interpolation of the equation through this transformation. The synthesis result of two variations of gain calculation logic with four pipeline stages is shown in Table 1. The optimization of the gain calculation module has a significant effect on improving the clock frequency and reducing the hardware area.

$$\frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} = \exp(2 \log |G_L| - \log |H_L + \lambda|) + \exp(2 \log |G_R| - \log |H_R + \lambda|) \quad (5)$$

$$W_L = -\exp(\log |G_L| - \log |H_L + \lambda|) \quad (6)$$

$$W_R = -\exp(\log |G_R| - \log |H_R + \lambda|) \quad (7)$$

5. C/C++ Based RTL Design Framework

5.1 C/C++ Based Design Flow

We made two major extensions to our C-based RTL design framework, C2RTL [14], as reported earlier. It is possible to input C++ code and describe not only modules but also the system level design. In addition, we described the entire GBT training system using that design framework.

In this framework, the C2RTL compiler tool automatically generates the RTL codes in Verilog-HDL, RTL-equivalent C (RTL-C) models that are cycle-accurate and bit-accurate, and RTL testbench codes that apply automatic test-vector validations of the entire system. The generated RTL-C models serve as the RTL verification platform in which the same software testbench code for the original C/C++ models are reused and RTL test-vectors are automatically generated. This mechanism is realized by our RTL expansion, pipelining, and optimization, using LLVM [15] as a compiler front-end and an intermediate language (Fig. 6). In addition, our C2RTL framework offers an enormous advantage in hardware design because a complete system-level hardware debugging can be performed on existing powerful software development platforms and tool sets.

Our RTL synthesis method is similar to high level synthesis (HLS), which is widely used in the field for generating RTL from C code, but takes a different approach. We describe “one cycle” behavior of the target with C/C++ codes and owing to this restriction, we can control the state transition of a cycle explicitly, define register level structure directly, and create a pipeline design as the initiation interval is always one clock cycle.

Fig. 6 C/C++ based RTL design framework (C2RTL) flow.

5.2 Source to Source Compiler

Unlike HLS, our framework provides a direct correspondence between the C/C++ code and the generated RTL. This important feature enables the designers to intuitively describe complex control paths as well as data paths by C/C++ codes. This property is realized mainly by two steps. The first step is to transform C/C++ codes into an inter-procedural call flow graph (I-CFG) by making a flat (non-hierarchical) and acyclic CFG by inlining all function calls under the target function and unrolling all loops. The second step is to extract data flow graph (DFG) by transforming each C statement into a static single assignment (SSA) form and replacing ϕ -nodes with multiplexers as the input condition for executing the control flow path. Then, each element of the extracted data flow graph is converted into the corresponding RTL expression.

We extended our C2RTL design framework to support C++ features such as class methods, virtual function, inheritance, and instantiation such that the reuse of the hardware component becomes extremely easy, including creating different versions of the prediction value equation and loss functions (gradient/hessian). Furthermore, supporting the template class, template function, constant expression functions ('constexpr') and other elements enables highly generic descriptions compared to standard HDL and greatly helps maintain the design parameters consistent across all components such as the feature dimensions and bit-precisions.

5.3 Control Path Description

When we describe a control path such as finite state machine, accessing the RAM or dealing with a bus transaction issue using a handshake procedure, we describe the explicit definition of the state registers and transition across the clock cycle boundary. Reading from and writing to the on-chip memory can be achieved using a normal C-array with a specified attribute annotation.

5.4 Data Path Description

When we describe the data path, such as a calculation of the gain without any state transition, we can determine an arbitrary number of pipeline stages for the target data flow description. When we define multiple pipeline stages for the data flow path, then optimizer of our tool distributes the gate delays to multiple pipeline stages and inserts registers between the clock boundaries automatically to optimize the clock frequency at the expense of the specified clock cycle latency limit.

Figure 7 shows an example code of the gain calculation data

Fig. 7 Example of data path description.

Fig. 8 Example of system level description.

path description. At the class definition part, we use template parameters to make a configurable module with the bit-width of the variables and the number of classes to classify, in addition, the bit-width for each variable is defined through an attribute annotation. At the RTL generation part, we describe each RTL module as a C/C++ function, instantiate all template parameters, and also determine the number of pipeline stages through an attribute annotation.

5.5 System Level Description

To describe the system level hardware logic, we annotate the specific attribute to the top function and required module are called inside this top function. An example of a top level module is shown in Figure 8. The first part of the code example focuses on the configurable parameters. Because all configuration parameters must be constant before instantiation of the hardware modules, the template technique and constant expression code of C++ are used as the parameter definitions and the conditional module instantiations are evaluated at the preprocessing of our C2RTL

Table 2 Configurable parameters.

Parameter	Range
LOSS_TYPE	Squared, LogReg
NUM_BINS	16, 32, 64, 128, 256
TREE_DEPTH	1–15
FRAC_BITS	1–16
PREDICT_BIT_WIDTH	12–32
GRADIENT_BIT_WIDTH	12–64
HISTOGRAM_BIT_WIDTH	24–64
GAIN_BIT_WIDTH	12–64
SAMPLES	100–1000000
SUB_SAMPLES	100–10000
FEATURE_DIMS	1–100
FETCH_FEATURE_DIMS	1–50
NUM_INTERLEAVE	2 ⁿ
CALC_LOGREGR_PIPELINES	4
CALC_GAIN_PIPELINES	2

compiler using the LLVM frontend.

For example, when we use the squared loss as the loss function for regression problems, the type of loss function is set to SQUARED, CalcGradSquare() is called, and the squared loss module is instantiated to calculate the slope of the loss function. Otherwise, CalcGradRegr() is called and the logistic regression loss module is instantiated instead. Both the CalcGradSquare() and CalcGradRege() functions use a class derived from the same base class to calculate the slope of the loss function. Any loss function can be employed in this way.

The configurable parameters used in our GBT experiments are listed in **Table 2**.

6. Implementation Results

6.1 Synthesis Result

We synthesized the generated RTL of a typical GBT training model, the parameters of which are denoted in **Table 3** and implemented on a Xilinx Kintex-7 UltraScale+ KCU116 evaluation board with the characteristics shown in **Table 4**. The synthesis results at a 250 MHz clock frequency are shown in **Table 5**. The ‘Number’ indicates the number of each instances according to the numbers of parallel features and sample interleaving. ‘Others’ includes the DMA controller to read from external memory and the host interface module through the bus. The values of this row are constant for any parameter configuration.

6.2 Evaluation Setup

In terms of accuracy and performance, we compared our FPGA implementation with a software implementation using the ‘xgboost 0.90’ library for both CPU only and GPU accelerated versions. We used an Intel Core i7-8700 CPU, and an NVIDIA GeForce-RTX2060 GPU with CUDA version 10.1. We used the xgboost library from the Python interface. For all of our experiments, we used the same parameters (Table 3) and variables (number of trees, learning rate, and regularization parameters) as used in our FPGA implementation and xgboost library. In addition, we used the same binned dataset for both the FPGA and software cases which are quantized by using the frequency-based algorithm such that the appearance frequencies are even. In the case of the FPGA, the processing time was measured after the embedded CPU transferred the training data to DRAM on the FPGA board.

Table 3 Configuration parameters for the experiments.

Parameter	Value
Num_Samples	10000, 100000
Num_Bins	256
Feature_Parallel	28
Num_Interleaves	8
Max_Depth	6
Frac_Bits	8
prediction_Bits	12
Gradient_Bits	22
Histogram_Bits	30

Table 4 Target FPGA.

Device	XCKU5P-2FFVB676E
Logic Elements	474600
Block RAM	34.9 Mbits
DSP slice	1824
Speed grade	–2

6.3 Accuracy Result

For the accuracy test, we use the binary classification dataset of Higgs [16] and Airline [17], the loss function of which is binary logistic regression and the evaluation function is a binary error. We randomly extracted 100,000 samples for both the Higgs and Airline datasets, and then used 80,000 samples as the training dataset and 20,000 samples as the validation (test) dataset. The validation error is obtained by applying the validation dataset to the tree created with the training dataset. We use an early-stop when the validation error does not improve further.

Figure 9 shows the curve of the training and validation errors with the Airline dataset where the number of bins is set to 16, 32, 64, 128, and 256. The bold curves correspond to the xgboost software implementation. Although the training error of the software implementation is constantly lower than our FPGA implementations, validation errors of the software implementation and FPGA implementations with 64 or more bins are quite similar on 200 or more trees. **Table 6** shows the validation errors with early-stop condition which also confirms our observations on the validation errors between the software implementation and FPGA implementations.

6.4 Performance Results

We compared the execution time for constructing 500 trees on a FPGA with a CPU-only version and a GPU-accelerated version of xgboost. The first case is 10,000 samples extracted from the Higgs (28 features) and Airline (13 features) datasets, respectively. In this case, all extracted feature data can be prefetched into an on-chip RAM concurrently, achieving a speed up of more than 17-times that of the GPU version (**Table 7**), if we increase the number of interleaves from 8 to 64, and synthesize on a larger capacity device such as a Vertex UltraScale+ VU5P at the same clock frequency (920K LUTs is needed), a speed-up of more than 70-times that of the GPU version is estimated through our cycle level simulation. Theoretically, our implementation has more advantages when a smaller bin number is used because the maximum gain search of each feature can be finished quickly, although in some cases, CPU processing with a small number of bins is advantageous possibly owing to the effect of the cache efficiency. In the second case of 100,000 samples (**Table 8**), our FPGA system needs to fetch samples 10 times during each histogram genera-

Table 5 Synthesis result at 250 MHz.

	Learning Control	Pointer memory	Prediction Memory	Calc Grad	Feature memory	Split	Histogram	Calc Gain	Calc Max-Gain	Others	Total
Number	1	8	8	8	224	8	224	28	28	1	-
LUTs	5122	6357	5615	11871	1807	1047	73099	29762	28787	1681	165148
Utilization	2.36%	2.93%	2.59%	5.47%	0.83%	0.48%	33.69%	13.72%	13.27%	0.77%	76.12%
Registers	3535	6264	904	168	2121	440	27764	10988	6476	1465	60125
Utilization	0.81%	1.44%	0.21%	0.04%	0.49%	0.10%	6.40%	2.53%	1.49%	0.34%	13.86%
BlockRAM	0	16	8	0	224	0	448	0	0	0	696
RAM (Bits)	0	11x13K	12x14K	0	8x9K	0	30x7K	0	0	0	593K
DSP	0	0	0	8	0	0	0	0	0	0	8

Fig. 9 Training and validation error curves with 16, 32, 64, 128, 256 bins and xgboost.

Table 6 Validation error with different number of bins.

Number of Bins	256	128	64	32	16	xgboost
Error						
Higgs	0.2755	0.2695	0.2715	0.275	0.275	0.2719
Airline	0.2715	0.271	0.271	0.300	0.344	0.2739

Table 7 Training time [seconds] for 10,000 samples.

Dataset	Bins	CPU	GPU	FPGA	Speed-up (vs. CPU)	Speed-up (vs. GPU)
Higgs	256	8.67	3.85	0.212	×40.9	×18.2
	64	4.41	3.66	0.0726	×38.3	×31.8
Airline	256	3.67	3.63	0.134	×17.3	×17.1
	64	2.95	3.85	0.0726	×25.7	×33.5

Table 8 Training time [seconds] for 100,000 samples.

Dataset	Bins	CPU	GPU	FPGA	Speed-up (vs. CPU)	Speed-up (vs. GPU)
Higgs	256	24.6	9.12	0.825	×29.8	×11.1
	64	20.24	9.16	0.727	×27.8	×12.6
Airline	256	15.54	8.8	0.825	×18.8	×10.7
	64	14.59	8.21	0.727	×20.1	×11.3

tion. This can disturb an additional speed-up particularly at a lower depth.

6.5 Energy Consumption

The estimated power of the FPGA device using the Xilinx Vivado tool was 3.29 W, including the power of the static and dynamic logic power, clock, and Block RAM, not including the off-chip memory. The estimated GPU power when using the Nvidia-smi utility was 95 W (max.160 W) at a utilization rate of 88%. A

Fig. 10 Comparison of number of LUTs with different amounts of sample interleaving for 7, 14, and 28 feature parallelism.

training power efficiency of 300 fold is achieved because it speeds up at least 10.7 times with 3.29 W of power when compared to a 95 W GPU.

6.6 Configuration Parameter Search

Some parameters have a trade-off; in particular, the buffer size for the feature data, and the amounts of sample interleaving and feature parallelism affect both the performance and hardware significantly. Therefore, a comparison between different parameters is needed to determine the optimal configuration parameters for a certain device.

We show the total number of LUT cells of each synthesis when the feature parallelism to 7, 14, and 28 and the sample interleaving to 1, 2, 4, 8, and 16, as shown in Fig. 10. All other parameters are fixed in this experiment. The number of LUTs is mostly linear to the degree of sample interleaving and feature parallelism. The number of training samples processed per cycle is shown in Fig. 11. The throughput is mostly linear to the degree of sample interleaving but not to the feature parallelism for both 10,000 and 100,000 samples, because feature parallelism accelerates only the split finding stage whereas sample interleaving accelerates all three stages.

The area efficiency curve (throughput/LUTs) is shown in Fig. 12. When the amount of feature parallelism is fixed, increasing the amount of sample interleaving always improves the area efficiency. By contrast, when the amount of sample interleaving is fixed, the area efficiency decreases when the amount of feature parallelism is increased. This indicates that it is more effective to increase the number of interleaves when the area is in a surplus. Thus, the strategy could to start with a small amount of feature parallelism and as much sample interleaving as possible,

Fig. 11 Throughput of 10,000 and 100,000 samples with sample interleaving (1, 2, 4, 8, 16) and feature parallelism (7, 14, 28).

Fig. 12 Area efficiency of 10,000 and 100,000 samples with sample interleaving (1, 2, 4, 8, 16) and feature parallelism (7, 14, 28).

and then increase the amount of feature parallelism depending on its available hardware area.

7. Conclusion

We proposed a pipeline-oriented hardware architecture, which uses feature prefetching with pointer memory structures, and ex-

ploits feature-level and sample interleaving parallelism. Some techniques are employed to speed-up and reduce the amount of hardware required for the numerical computations.

To realize this hardware architecture, we made extensive use of our C2RTL design framework, which converts C/C++ descriptions directly into system-level RTL descriptions, enabling high-quality descriptions of both the data-path and control-path, and at the same time enabling a flexible configuration to meet the system requirements. Experimental result shows that an implementation to the small FPGA device achieves a 11–33 times speed-up compared to the GPU approach, and more than a 70-times speed-up is possible by simply changing the parameters when we use a large capacity FPGA device.

References

- [1] Lundberg, S.M., Erion, G.G. and Lee, S.-I.: Consistent individualized feature attribution for tree ensembles, *arXiv preprint arXiv:1802.03888* (2018).
- [2] Puurula, A., Read, J. and Bifet, A.: Kaggle LSHTC4 winning solution, *arXiv preprint arXiv:1405.0546* (2014).
- [3] Taieb, S.B. and Hyndman, R.J.: A gradient boosting approach to the Kaggle load forecasting competition, *International Journal of Forecasting*, Vol.30, No.2, pp.382–394 (2014).
- [4] Chapelle, O. and Chang, Y.: Yahoo! learning to rank challenge overview, *Proc. Learning to Rank Challenge*, pp.1–24 (2011).
- [5] Mohan, A., Chen, Z. and Weinberger, K.: Web-search ranking with initialized gradient boosted regression trees, *Proc. Learning to Rank Challenge*, pp.77–89 (2011).
- [6] Ke, G., Meng, Q., Finley, T., Wang, T., Chen, W., Ma, W., Ye, Q. and Liu, T.-Y.: Lightgbm: A highly efficient gradient boosting decision tree, *Advances in Neural Information Processing Systems*, pp.3146–3154 (2017).
- [7] Mitchell, R., Adinets, A., Rao, T. and Frank, E.: Xgboost: Scalable GPU accelerated learning, *arXiv preprint arXiv:1806.11248* (2018).
- [8] Wen, Z., He, B., Kotagiri, R., Lu, S. and Shi, J.: Efficient gradient boosted decision tree training on GPUs, *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp.234–243, IEEE (2018).
- [9] Lo, W.-T., Chang, Y.-S., Sheu, R.-K., Chiu, C.-C. and Yuan, S.-M.: CUDT: A CUDA based decision tree algorithm, *The Scientific World Journal*, Vol.2014 (2014).
- [10] Wen, Z., Shi, J., He, B., Li, Q. and Chen, J.: ThunderGBM: Fast GB-DTs and Random Forests on GPUs (2019).
- [11] Tanaka, T., Kasahara, R. and Kobayashi, D.: Efficient logic architecture in training gradient boosting decision tree for high-performance and edge computing, *arXiv preprint arXiv:1812.08295* (2018).
- [12] Chen, T. and Guestrin, C.: Xgboost: A scalable tree boosting system, *Proc. 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp.785–794 (2016).
- [13] Amin, H., Curtis, K.M. and Hayes-Gill, B.R.: Piecewise linear approximation applied to nonlinear function of a neural network, *IEE Proceedings-Circuits, Devices and Systems*, Vol.144, No.6, pp.313–317 (1997).
- [14] Isshiki, T., Date, K., Kugimiya, D., Li, D. and Kunieda, H.: C-based RTL design framework for processor and hardware-IP synthesis, *Proc. SASIMI*, pp.40–45 (2015).
- [15] Lattner, C. and Adve, V.: LLVM: A compilation framework for life-long program analysis & optimization, *International Symposium on Code Generation and Optimization, CGO 2004*, pp.75–86, IEEE (2004).
- [16] Baldi, P., Sadowski, P. and Whiteson, D.: Searching for exotic particles in high-energy physics with deep learning, *Nature Communications*, Vol.5, No.1, pp.1–9 (2014).
- [17] Airline dataset, available from (http://kt.ijs.si/elena_ikonmovska/data.html).

Tamon Sadasue received his B.E degree from Tokyo Institute of Technology in 2003 and M.E. degree from Tokyo University in 2005. He joined Ricoh in 2005, and has been engaged in research of 2D and 3D image processing. His current primary research interests are high efficient design methodology of logic circuit and

hardware acceleration of machine learning.

Takuya Tanaka was born in 1988. He received his M.E. degree from The University of Electro-Communications in 2013. He joined Ricoh in 2013, where he was engaged in research on machine learning for fintech and image recognition for visual inspection services. He is currently at Innovation/R&D Division in

Ricoh.

Ryosuke Kasahara was born in 1980. He received his M.E. and Ph.D. from Tohoku University in 2004 and 2019, respectively. He has been engaged in research on signal processing, image processing, and machine learning in Ricoh since 2004.

Arief Darmawan received his Bachelor degree in 2016 from Universitas Gadjah Mada, Indonesia and Master degree in 2020 from Tokyo Institute of Technology. His research interest is hardware accelerator design for machine learning application.

Tsuyoshi Isshiki has received his B.E. and M.E. degree in Electrical and Electronics Engineering from Tokyo Institute of Technology in 1990 and 1992, respectively, and Ph.D. degree in Computer Engineering from University of California at Santa Cruz in 1996. He is currently a Professor at Department of Information

and Communications Engineering, Tokyo Institute of Technology. His research interests include design methodologies for System-on-Chip architectures, application-specific processors, image processing systems, MPSoC, and fingerprint authentication algorithms. He is a member of IEEE CAS, IPSJ, and IEICE.

(Recommended by Associate Editor: *Salita Sombatsiri*)