An FPGA Realization of a Random Forest with \( k \)-Means Clustering Using a High-Level Synthesis Design

**SUMMARY** A random forest (RF) is a kind of ensemble machine learning algorithm used for a classification and a regression. It consists of multiple decision trees that are built from randomly sampled data. The RF has a simple, fast learning, and identification capability compared with other machine learning algorithms. It is widely used for application to various recognition systems. Since it is necessary to un-balanced trace for each tree and requires communication for all the ones, the random forest is not suitable in SIMD architectures such as GPUs. Although the accelerators using the FPGA have been proposed, such implementations were based on HDL design. Thus, they required longer design time than the software based realizations. In the previous work, we showed the high-level synthesis design of the RF including the fully pipelined architecture and the all-to-all communication. In this paper, to further reduce the amount of hardware, we use \( k \)-means clustering to share comparators of the branch nodes on the decision tree. Also, we develop the \textit{krange} tool flow, which generates the bitstream with a few number of hyper parameters. Since the proposed tool flow is based on the high-level synthesis design, we can obtain the high performance RF with short design time compared with the conventional HDL design. We implemented the RF on the Xilinx Inc. ZC702 evaluation board. Compared with the CPU (Intel Xeon (R) E5607 Processor) and the GPU (NVidia Geforce Titan) implementations, as for the performance, the FPGA realization was 8.4 times faster than the CPU one, and it was 62.8 times faster than the GPU one. As for the power consumption efficiency, the FPGA realization was 7.8 times better than the CPU one, and it was 385.9 times better than the GPU one.

**key words:** machine learning, random forest, \( k \)-means clustering, FPGA

**1. Introduction**

**1.1 Acceleration of the Random Forest (RF)**

A decision tree is a popular method for various machine learning tasks. When a tree is grown very deep to learn highly irregular patterns, it overfits training sets. In that case, it has a low bias, while it has very high variance. Since the decision tree partitions the data set with a single feature variable, it often misclassifies out-liar labels. A random forest (RF) [5] is an ensemble learning method for a classification and a regression, and it consists of multiple decision trees. As for training, each decision tree is built by different (randomized) samples from the same training set. Since the RF uses training feature variables selected at random sampled, decision trees with low correlation are built. As a result, it improves accuracy and versatility compared with a single decision tree based classification.

The RF is widely used for a classification. For example, they are a key point matching [19], a segmentation [2], a pedestrian detection [7], [10], a human pose estimation [22], a face direction estimation [8], and an IP address search for the Internet [12]. These applications are demanded to be recognized in real time. However, since the classification speed in the CPU is too slow, the hardware acceleration is necessary. Also, since it is often used in embedded systems, the low power consumption is desired. However, a single instruction multiple data (SIMD) architecture, typified by GPU, is not suitable for the RF with three reasons as follows:

1. **Higher precision:** To evaluate decision trees in the RF, each node in the decision tree can be evaluated by \textit{if-then-else} statements. A conditional expression in the \textit{if-then-else} statement is a comparison an input with a constant value, which is represented by a floating point representation. Although the GPU supports a double precision floating point, such high precision is not required for the RF classification. Therefore, a high-precision arithmetic circuit is inefficient both amount of hardware and power consumption.

2. **Uniformly processing (CUDA) cores:** The GPU runs the SIMD operations, that is, having a large amount of uniformly processing core. These cores are specialized in data parallel computation. However, the RF consists of decision trees with a different size, that causes an unbalanced computation. Since a warp divergent is frequently occured, computation time would be bound by the decision tree which has the longest path.

3. **Higher cost for the all-to-all communication:** The GPU can be performed at a relatively high speed communication between near processing cores with the same local memory, while its communication penalty is large for the all-to-all communication. Since an RF requires the whole of the majority detection after the evaluation of all the decision trees, the all-to-all communication would always occur.
1.2 FPGA Realizations of the RF

Since the FPGA is possible to configure a dedicated all-to-all communication circuit, heterogeneous cores for different size of a decision tree, and an appropriate variable bit length circuit. Thus, the FPGAs are suitable to accelerate the RF. Becker et al. used decision trees to accelerate object tracking. They focused on heavily parallelizing the classification, and converted the input data into a dedicated representation in the FPGA [4]. Essen et al. showed a pipelined architecture and a single-instruction multiple thread (SIMT) algorithm for the RF on FPGA. Also, they compared the FPGA based implementation with the multi-core CPU and the GPU [9]. Oberg et al. implemented the RF on the FPGA with the Kinect depth-image sensor for the Forest Fire pixel classification algorithm [17]. However, the conventional realizations are designed by the RTL description. Compared with the software-based design, it takes an enormous amount of development time [1]. As for the RF design, since its structure is completely different for each dataset, it is not practical to tune the architecture by the RTL description.

In recent years, to reduce the development time, many of high-level synthesis tools have been proposed. A typical high-level synthesis supports the C/C++ codes as input design. Especially, the Intel SDK for OpenCL [11] and the Xilinx SDSoC [24] are a system level design tool which includes system level profiling, automated software acceleration in programmable logic, automated system connectivity generation, and libraries to speed programming. Therefore, it can accelerate the target application with short development time.

In this paper, we accelerate the RF on the FPGA. An acceleration method of the RF for training using the FPGA has been already proposed [16]. In this paper, we assume that the learning is done by off-line. The paper compares with the software-based design such as a CPU and a GPU as for the classification time and power consumption efficiency.

1.3 Contributions of the Paper

In the previous contributions [15] were as follows:

1. We proposed two acceleration methods for the pre-learned RF using a high-level synthesis tool as follows; using an appropriate bit fixed point representation instead of a floating point one; and realizing a fully pipelined architecture including the all-to-all communication circuit.

2. We compared with the software based realization, such as the CPU and the GPU. Compared with the GPU realization, as for the performance, the FPGA realization was 10.7 times faster than the GPU one, and it was 14.0 times faster than the CPU one. As for the power consumption efficiency, the FPGA realization was 61.3 times better than the GPU one, and it was 12.1 times better than the CPU one.

3. We opened the Python based code generation flow (RF2AOC), which focuses on the Altera’s FPGA tool.

Additionally, in the paper, new contributions were as follows:

1. We included the $k$-means clustering, which merges the multiplexers into a single one. As a result, the amount of area was reduced, while it kept the baseline accuracy.

2. We compared with the software based realization, such as the CPU and the GPU. Compared with the GPU realization, as for the performance, the FPGA realization was 8.4 times faster than the CPU one, and it was 62.8 times faster than the GPU one. As for the power consumption efficiency, the FPGA realization was 385.9 times better than the GPU one, and it was 7.8 times better than the CPU one.

3. We released the Python based code generation flow including the $k$-means ($k$range), which focuses on the Xilinx SDSoC.

This paper is an updated version of the previous publication [15].

The rest of the paper is organized as follows: Sect. 2 compares the GPU computation model with the FPGA one; Sect. 3 introduces the random forest (RF); Sect. 4 shows an acceleration techniques used in the previous work; Sect. 5 proposes a comparator sharing by $k$-means clustering; Sect. 6 shows a proposed tool flow, which is called $k$range; Sect. 7 shows the experimental results; and Sect. 8 concludes the paper.

2. High Performance and Short Time Design Using the FPGA

2.1 Comparison of Computation Model

In recent years, the Altera Corp. has promoted the Altera SDK for OpenCL for the FPGA development environment, and the Xilinx Inc. also has released the SDSoC. These tools include system level profiling, automated software acceleration in programmable logic, automated system connectivity generation, and libraries to speed programming. Although they provide a familiar C/C++ application development experience, when we directly applied the GPU programming model to them, since the target computation models are different, it is hard to accelerate the application even if we used such a system level tool.

Here, we explain the programming model for the FPGA fitted system and the GPU fitted one. Figure 1 shows an architecture model for the GPU, while Fig. 2 shows that for the FPGA. For the GPUs, it runs the threads in the工作组 to the data parallel model by using a large number of CUDA cores and wide band data transfer memory such as DDR5 off-chip memories. On the other hand, for the
FPGA, since data communication bandwidth, that is, that for off-chip memories are narrow, it tends to configure the pipeline model in the work-group. When the pipeline stall is free, communications with the off-chip memory are only input and output of the pipeline. Therefore, the FPGA can realize a high-throughput operation even if it has a narrow band to the off-chip memories. Fortunately, since the FPGA has more on-chip memories than the GPU, data transfer between the pipeline stages is often done on the FPGA. The modern system design tools for the FPGA supplies a channel to make an on-chip communication between pipeline stages. Furthermore, another advantage of the FPGA is that it is possible to realize a customized pipeline stage (in other words, it can realize a heterogeneous core) in parallel. When the latency of each parallel operation is different, its computation time for the GPU would be bound by the longest one. On the other hand, the hardware resources in the FPGA are appropriately distributed, it is possible to realize a heterogeneous parallel architecture with uniform latency.

2.2 Short Time Design Using the High-Level Tool

Another feature of the system design tool is a short time design compared with the conventional HDL design. The modern system level design, such as an Intel SDK for OpenCL and a Xilinx SDSoC, supplies the board support package (BSP) for recommended FPGA boards. The BSP prepares the IP cores and external memories to bridge host program and the kernel program on the host processor. For the conventional FPGA design, since the programmer designed them in the RTL description, it could not respond to frequent changes in long term design. By using such a system level design tool, it can generate the configuration data that automatically connects with the user program. Therefore, since it is possible to remarkably reduce design time, the programmer can concentrate on tuning architectures. The remaining problem for the RF design on the FPGA is the C/C++ code refactoring, which consumes design time. In the paper, we propose the krange tool flow, which automatically generates optimized C/C++ code in order to reduce the design time.

3. Random Forest (RF)

A decision tree is a popular method for various machine learning tasks. When a tree is grown very deep to learn highly irregular patterns, it overfits training sets. In that case, it has a low bias, while it has very high variance. Figure 4 shows an example of a decision tree which classifies a data set shown in Fig. 3. In Fig. 4, \( X_i \) denotes a feature variable for the dataset, and \( C_i \) denotes a label. Since the decision tree partitions the data set with a single feature variable, it often misclassifies out-lier labels.

A random forest (RF) is an ensemble learning method for a classification and a regression, and it consists of multiple decision trees. At learning, each decision tree is built by different (randomized) sub-sampling data from the same training set in order to reduce the variance. Figure 5 shows an example of the RF, which consists of \( b \) decision trees and a voter. First, decision trees branch according to given feature variables. Then, they output the matched label. Next, the voter performs a majority decision of the labels from decision trees. Finally, it detects the most frequent label as a classification result. Since the RF uses training feature variables selected at random sampled, decision trees with low correlation are built. As a result, it improves accuracy and versatility.

Followings are an algorithm to built the RF from a given dataset.

```
DDRA/DDR5

Wide Band

D1 D2 D3 D4 D5

↓ ↓ ↓ ↓ ↓

PE PE PE PE PE

↓ ↓ ↓ ↓ ↓

Fig. 1 Computation model for the GPU.

DDR3

↓ Narrow

PE

D1 D2 D3 D4

↓ ↓ ↓ ↓

PE PE PE

↓ t=1 t=2 t=3 t=4 t=5 t=6 t=7

Fig. 2 Computation model for the FPGA.
```
JINGUI et al.: AN FPGA REALIZATION OF A RANDOM FOREST WITH K-MEANS CLUSTERING USING A HIGH-LEVEL SYNTHESIS DESIGN

Figure 5: Example of a random forest.

Algorithm 3.1: 1. Randomly selects $b$ sub-samples from given dataset (bootstrap sampling)
2. Learns $b$ decision trees from $b$ sub-samples
3. Creates a node to reach the specified number of nodes $N_{\text{min}}$

3.1. Selects $r$ samples at random
3.2. Computes a constant values for if-then-else statements in a node, which classifies sub sampling data
4. Terminate

The advantages of the RF are shown as follows [5]:
1. Classification accuracy is high, and it operates correctly even if the feature variables are from several hundreds to thousands
2. It is possible to estimate the importance of the feature variables for each label variable
3. It effectively works with dataset even if it lacks several feature variables
4. The number of individual error are maintained even in unbalanced dataset

On the other hand, the disadvantages are follows:
1. Too deep decision trees fall into over fitting
2. Classification accuracy is low with a small number of learning data

In addition, classification accuracy is greatly affected by hyper parameters. By using a grid search algorithm and encourage parameters $^\dagger$, the RF can be built with relatively appropriate hyper parameters.

4. Acceleration Techniques Used in the Previous Work

4.1 Fixed Point Representation

As shown in Fig. 1 and 2, bandwidth between the off-chip memory and the kernel on the FPGA become a bottleneck. Each node in a decision tree can be expressed by if-then-else statements. For many of random forest software libraries, a conditional expression in the if statement is a comparison a feature variable with a constant value which is represented by a 32-bit floating point representation. In this paper, we use an $n$-bit signed fixed point representation instead of 32-bit floating point one. According to the bit width, a fixed point representation lacks accuracy compared with a floating point one. Thus, a fixed-point representation may cause misclassification. However, since it compresses the bandwidth of the off-chip memory to $\frac{n}{32}$, it can accelerate the classification. Furthermore, since multiplexer trees for a fixed point are simplified, they are faster and smaller than floating point based multiplexers.

4.2 Pipeline Stages by a Loop Unrolling

In the paper, we connect decision trees in series to form a deep pipeline with on-chip memory. It increases the system throughput. Also, we realize the voter by the pipeline circuit. Figure 7 shows an example of decision trees, and
their hardware realization by a multiplexer tree. When the decision trees are written by a `for` statement, as shown in Fig. 8, the high-level synthesis tool sequentially traces decision trees by a shared multiplexer circuit. On the other hand, we can increase the throughput by using an `#pragma unroll` which expands a sequential circuit to a pipelined one. Figure 9 shows a pipeline circuit with an `#pragma unroll`. In the pipeline circuit, registers and voters are inserted between the multiplexer trees. In that case, the number of registers and the memories tend to be increased, and they are realized by the on-chip memories. Since communication with the off-chip memory does not occur, it can accelerate throughput.

5. Comparator Sharing by \( k \)-Means Clustering

5.1 Hardware Reduction by Comparator Sharing

A typical approach to reducing the amount of hardware is a resource sharing. In the RF, a comparison of the input variable with the threshold is required in each node of a decision tree. They can be shared by summarizing comparisons which have similar thresholds for certain input variables. Each node of the decision tree compares the input variable with the constant threshold. In other words, it divides the feature space by the constant threshold. Since to summarize similar comparisons is the same as roughening the division of feature space, it is expected that the classification accuracy may be worse. Thus, an appropriate sharing is necessary to find small hardware with the classification accuracy. In this paper, we share similar comparisons using the \( k \)-means method, which is a kind of clustering algorithm. Figure 10 shows an example of \( k \)-means clustering. In other words, it reduces the number of thresholds for feature space partitioning. By applying \( k \)-means clustering, comparators can be merged into single one. Figure 11 shows an example of comparator sharing by \( k \)-means clustering.

5.2 \( k \)-Means Clustering

A \( k \)-means clustering \cite{14} is one of the simplest unsupervised learning algorithms. The procedure classifies a simple and easy to classify a given data set by using a certain number of clusters (assume \( k \) clusters) following a simple and

![Fig. 8](image_url)  
**Fig. 8** Sequential realization.

![Fig. 9](image_url)  
**Fig. 9** Pipeline realization with an `unroll` pragma.

![Fig. 10](image_url)  
**Fig. 10** Example of \( k \)-means clustering.

![Fig. 11](image_url)  
**Fig. 11** Example of comparator sharing by \( k \)-means clustering.
easy ways. The main idea is to define a center value for each cluster. These $k$ centroid values should be placed in a clever way, since different location causes a different result. Thus, the better choice is to place them as much as possible far away from each other. The next step is to take each point belonging to a given data set and associate it to the nearest centroid. When no point is pending, the first step is completed and an early grouping is done. At this point, re-calculation for $k$ new centroids is performed as barycenters of the clusters resulting from the previous step. After these $k$ new centroids are obtained, a new binding has to be done between the same data set points and the nearest new centroid. $k$ centroids are changed their location step by step until no more changes are done. In other words, centroids do not move any more. Finally, this algorithm aims at minimizing an objective function. Let $n$ be the number of data points in the cluster. In this case, a squared error function $F_{\text{err}}$ is defined as follows:

$$F_{\text{err}} = \sum_{j=1}^{k} \sum_{i=1}^{n} ||x_i^{(j)} - c_j||^2,$$

where $||x_i^{(j)} - c_j||^2$ is a distance measure between a data point $x_i^{(j)}$ and the cluster center $c_j$. It indicates the distance of the $n$ data points from their respective cluster centers.

The algorithm consists of the following steps:

**Algorithm 5.2:**
1. Place $k$ points into the space represented by the objects that are being clustered. These points represent initial group centroids.
2. Assign each object to the group that has the closest centroid.
3. When all objects have been assigned, recalculate the positions of the $k$ centroids.
4. Repeat Steps 2 and 3 until the centroids are no longer changed. This produces a separation of the objects into groups from which the metric to be minimized can be calculated.

An advantage of the $k$-means method is known that its algorithm is simple and operates at high speed. The result of clustering depends on the random initial value for centroids.

[3] proposed a devising the initial allocation for centroids. Since it is necessary to give the number of clusters $k$, it uses other indicators in order to select the optimum number of clusters. An estimation method for the optimum number of clusters using the $k$ average method [20] has been proposed.

By using the $k$-means clustering, a similar comparison operation of the random forest is shared. The algorithm for the comparator sharing RF is shown as follows:

**Algorithm 5.3:**
1. At the training, thresholds of nodes are collected for each feature variable.
2. The thresholds recorded for each feature variable are clustered by the $k$-means method and classified into $k$ clusters, respectively.
3. At the inference, each centroid is used for a comparison with the corresponding input feature variable.

In the RF, each node consists of a comparator and a multiplexer. Since the centroid is used as the threshold for branching of the multiplexer, almost initial comparators can be shared. Therefore, it is possible to drastically reduce the comparator in the random forest.

5.3 Proposed Tool Flow

Figure 12 shows the proposed tool flow, which uses **krange (k-means based random forest generator)**. First, we use the scikit-learn software [21] to learn the RF from given dataset. Note that, we find the optimum hyper parameter set by a grid-search algorithm. Then, we shared the threshold by $k$-means clustering with hyper parameters, which is the number of clusters. Next, we generate the host code and the kernel code for the high-level synthesis tool for the FPGA. The generated codes are converted into the bit stream file by using the logic synthesis tool. Since the proposed tool flow automatically generates the bit stream from given dataset, we can concentrate the parameter tuning to accelerate the RF.
6. Experimental Results

6.1 Implementation Environment

We implement the UC Irvine machine learning repository [23] to a Xilinx Inc. ZC702 evaluation board, which has a Xilinx Corp. Zynq7020 (53,200 LUTs, 140 36Kb BRAMs, 220 DSP blocks). To generate an executable code, we used the proposed tool flow shown in Fig. 12. For the host PC, we used the Intel’s Xeon (R) E5607 Processor (2.26GHz, 4 cores) with 32GB DDR3 off-chip memory, and Ubuntu 14.04 LTS (64 bit version) operating system. Table 1 shows the used dataset and parameters for them. Note that, to find the optimum parameter set, we used a grid search algorithm which is available in scikit-learn. We showed the dynamic power consumption. To measure it, we measured both the static power consumption and the total power consumption to compute random test vectors. Then, we obtained the dynamic one.

From the previous work [15], since the 14 bit fixed point precision caused no classification error degradation, we used such custom precision to the RF implementation. In the implementation, we inserted `pragma unroll` in the top of each loop, thus the number of pipeline stages is equal to the depth for each decision tree. In Table 1, we showed the pipeline stages as (PS). Also, we showed the number of clusters $k$ used in the experiment.

6.2 Compared with the Conventional Method

We set the number of clusters $k$ to 1, 2, 4, and 8, respectively. Then we measured the number of comparators (thresholds) and the classification accuracy. We compared the proposed comparator sharing with the conventional realization [15], which did not share the comparators. Figure 13 compares the number of LUTs which is a bottleneck of the RF implementation and Fig. 14 compares the number of flip-flops (FFs). Figure 15 compares the classification error rate. As shown in Fig. 13 and 15, there is a trade-off between the hardware consumption and the classification accuracy. It is dependent on the variation of dataset. In Fig. 15, even if we decreased $k$, the classification error for Dermatology did not increase. Since its feature values exist around similar values, the clustered value took almost the same value. For another dataset, when $k = 2$, their error rates are considerable, and when $k = 4$, those rates are slightly increased. Thus, for all cases, we think that $k = 4$ is practical value. Above discussions showed that, we should carefully consider the error rate when we design the RF on the FPGA using a clustering method, since it is depended on its dataset. From the experiment, the hardware resource usage is reduced by 41% at maximum with 1% error reduction of the baseline accuracy, while it is reduced by 64% at one with 5% or less. Thus, the $k$-means clustering efficiently reduced the amount of hardware.

6.3 Compared with Other Platforms

As for the lookups per second (LPS) and the power consumption efficiency (LPS/W), we compared the FPGA with

| Dataset | Hyper Parameters | PS |
|---------|------------------|----|
| Arrhythmia | 452 | 279 | 16 | 35 | 20 | 20 | 4 | 4 | 20 |
| Dermatology | 366 | 33 | 6 | 30 | 5 | 7 | 1 | 5 |
| Ionosphere | 351 | 34 | 2 | 25 | 15 | 10 | 4 | 15 |
| Iris | 150 | 4 | 3 | 50 | 20 | 2 | 4 | 20 |

Table 1 Dataset used in the experiment.

Fig. 13 Comparison of the number of LUTs.

Fig. 14 Comparison of the number of flip-flops (FFs).

Fig. 15 Comparison of classification error (%).
the CPU and the GPU. Figure 16 shows the execution flow used in the experiment. As for the CPU platform, we used the Intel’s Xeon (R) E5607 Processor (2.26GHz, 4 cores) with 32GB DDR3 off-chip memory, and Ubuntu 14.04 LTS (64 bit version) operating system. To generate the executable code, first, we generated the RF by the scikit-learn with the same parameters shown in Table 1. Then, we convert the RF to C-codes by Cython [6], and compiled to executable code by gcc compiler. As for the GPU platform, we used the NVidia Geforce Titan (876 MHz, 2,496 CUDA cores, and 6GB DDR5 off-chip memory) with the same processor and the main memory running on the Ubuntu 14.04 LTS. To generate the executable code, first, we generated the RF by the scikit-learn, then, we used the CUDA Tree (CUDA T) [13] to generate the executable code. To measure the LPS, we used 1,000 random test vectors, while to measure the power consumption excluding the idle power, we inserted the power measurement between the host PC and the power source. As for the FPGA realization, from above experiments, we set appropriate numbers of bits (14 bit fixed point precision) and unrolls. On the other hand, for both the CPU and the GPU, 32 bit floating point precision. Since the FPGA only can realize a custom bit-length precision to realize a high-performance circuit, it is an advantage to use the FPGA.

Comparison with the previous implementation in [15], the present result is worse. Since the previous implementation used the high-end FPGAs in order to use at the data center, while this implementation used the low-end FPGAs in order to the embedded system for a low-cost system. Table 2 compared with the CPU and the GPU realizations. Note that, we used the best k which reduced maximumly hardware resource and its classification error reduction was around 1% of the baseline accuracy. As shown in Table 2, compared with the CPU, the FPGA realization is 8.4 times faster and 7.8 times better power efficiency. Compared with the GPU, it is 62.8 times faster and 385.9 times better power efficiency. Since the RF requires heterogeneous if-then-else statements, the conventional homogeneous architectures are not suitable for such application. Only the FPGA can configure custom pipelined architecture for branch operations in the RF, thus, it achieved higher performance and lower power consumption.

7. Conclusion

This paper showed the acceleration method for the RF on the FPGA. To accelerate the RF, we proposed the fully pipelined architecture including all-to-all communication circuit. It increased the memory bandwidth using on-chip memories on the FPGA. To further improve the RF for the FPGA realization, we used k-means clustering to share the comparator of the decision tree on the RF. We also developed the krang tool flow, which generates the bitstream with only a few hyper parameters. We implemented UC Irvine machine learning repository dataset on the Xilinx Inc. ZC702 evaluation board. Compared with the CPU and the GPU realizations, as for the LPS, the FPGA realization was 8.4 times faster than the CPU one, and it was 62.8 times faster than the GPU one. As for the LPS per power consumption, the FPGA realization was 7.8 times better than the GPU one, and it was 385.9 times better than the GPU one.

Acknowledgments

This research is supported in part by the Grants in Aid for Scientific Research of JSPS, and an Accelerated Innovation Research Initiative Turning Top Science and Ideas into High-Impact Values program (ACCEL) of JST.

References

[1] M.S. Abdelfattah, A. Hagiescu, and D. Singh, “Gzip on a chip: high performance lossless data compression on FPGAs using OpenCL,” Int'l Workshop on OpenCL 2013 and 2014 (IWOCL), no.4, 2014.
[2] Y. Amit and D. Geman, “Shape quantization and recognition with randomized trees,” Neural Computation, vol.9, no.7. Oct. pp.1545–1588, 1997.
[3] D. Arthur and S. Vassilvitskii, “k-means++: The advantages of careful seeding,” Proc. eighteenth annual ACM-SIAM symposium on Discrete algorithms, Society for Industrial and Applied Mathematics, pp.1027–1035, 2007.
[4] T. Becker, Q. Liu, W. Luk, G. Nebehay, and R. Pflugfelder, “Hardware-accelerated object racking,” Computer Vision on Low-Power Reconfigurable Architectures Workshop, pp.1–2, 2011.
[5] L. Breiman, “Random forests,” Machine Learning, vol.45, no.1, pp.5–32, 2001.
[6] “Cython: C-Extensions for Python,” http://cython.org/
[7] D. Tang, Y. Liu, and T.K. Kim, “Fast pedestrian detection by cascaded random forest with dominant orientation templates,” BMVC, pp.58.1–58.11, 2012.
[8] M. Dantone, J. Gall, G. Fanelli, and L.V. Gool, “Real-time facial
feature detection using conditional regression forests,” IEEE Conf. on Comput. Vis. Pattern Recognit. (CVPR), pp.2578–2585, 2012.

[9] B. V. Essen, C. Macareag, M. Gokhale, and R. Prenger, “Accelerating a Random Forest Classifier: Multi-Core, GP-GPU, or FPGA?,” IEEE Annual Int’l Symp. on Field-Programmable Custom Computing Machines (FCCM), pp.232–239, 2012.

[10] S. Hinterstoisser, V. Lepetit, S. Ilic, P. Fua, and N. Navab, “Dominant orientation templates for real-time detection of texture-less objects,” CVPR, 2010.

[11] Intel Corp., “Intel SDK for OpenCL,” http://www.intel.com/

[12] H. Le, W. Jiang, and V.K. Prasanna, “A SRAM-based architecture for Trie-based IP lookup using FPGA,” FCCM, pp.33–42, 2008.

[13] W.T. Lo, Y.S. Chang, R.K. Sheu, C.C. Chiu, and S.M. Yuan, “CUDT: A CUDA based decision tree algorithm,” The Scientific World Journal, vol.2014, no.745640, pp.1–12, 2014.

[14] J. MacQueen, “Some methods for classification and analysis of multivariate observations,” Proc. Fifth Berkeley Symp. on Math. Statist. and Prob. Univ. of Calif., pp.281–297, 1967.

[15] H. Nakahara, A. Jingui, T. Fujii, and S. Sato, “An acceleration of a random forest classification using altera SDK for OpenCL,” The Int’l Conf. on Field-Programmable Technology (FPT), pp.285–288, 2016.

[16] R. Narayanan, D. Honbo, G. Memik, Al Choudhary, and J. Zambreno, “An FPGA implementation of decision tree classification,” Design Automation and Test Europe Conference (DATE), pp.1–6, 2007.

[17] J. Oberg, K. Eguro, R. Bittner, and A. Forin, “Random decision tree body part recognition using PDFs,” The Int’l Conf. on Field Programmable Logic and Applications (FPL), pp.330–337, 2012.

[18] The OpenCL specification, version 1.2,” http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf

[19] M. Ozuysal, M. Calonder, V. Lepetit, and P. Fua, “Fast keypoint recognition using random ferns,” IEEE Trans. Pattern Anal. Mach. Intell., vol.32, no.3, pp.448–461, 2010.

[20] D. Pelleg and A.W. Moore, “X-means: Extending k-means with Efficient Estimation of the Number of Clusters,” ICML, vol.1, 2000.

[21] “Scikit-learn: Machine Learning in Python,” http://scikit-learn.org/stable/

[22] J. Shotton, A. Fitzgibbon, M. Cook, T. Sharp, M. Finocchio, R. Moore, A. Kipman, and A. Blake, “Real-time human pose recognition in parts from single depth images,” CVPR, vol.2, pp.1297–1304, 2011.

[23] “UCI Machine Learning Repository,” http://archive.ics.uci.edu/ml/datasets.html

[24] Xilinx Inc., “Xilinx SDSoC development environment,” http://www.xilinx.com/

Shinpei Sato received the B.S., M.S., and Ph.D. degrees in engineering from Tokyo Institute of Technology (Tokyo Tech), in 2007, 2009, and 2014. He is currently an Assistant Professor with the Department of Information and Communications Engineering of Tokyo Tech. From 2014 to 2016, he worked in High performance computing area as a post doctoral researcher, where he investigated an application performance analysis/tuning method. His current research interests include approximate computing realization by architecture design and circuit design and their applications.

Hiroki Nakahara received the B.E., M.E., and Ph.D. degrees in computer science from Kyushu Institute of Technology, Fukuoka, Japan, in 2003, 2005, and 2007, respectively. He has held research/faculty positions at Kyushu Institute of Technology, Iizuka, Japan, Kagoshima University, Kagoshima, Japan, and Ehime University, Ehime, Japan. Now, he is an associate professor at Tokyo Institute of Technology, Japan. He was the Workshop Chairman for the International Workshop on Post-Binary ULSI Systems (ULSIWS) in 2014, 2015, 2016, and 2017, respectively. He served the Program Chairman for the International Symposium on 8th Highly-Efficient Accelerators and Reconfigurable Technologies (HEART) in 2017. He received the 8th IEEE/ACM MEMOCODE Design Contest 1st Place Award in 2010, the SASIMI Outstanding Paper Award in 2010, IPSJ Yamashita SIG Research Award in 2011, the 11st FIT Funai Best Paper Award in 2012, the 7th IEEE MCSoC-13 Best Paper Award in 2013, and the ISMVL2013 Kenneth C. Smith Early Career Award in 2014, respectively. His research interests include logic synthesis, reconfigurable architecture, digital signal processing, embedded systems, and machine learning. He is a member of the IEEE, the ACM, and the IEICE.

Akira Jingui received the B.S. degrees from Tokyo Institute of Technology (Tokyo Tech), in 2017. His current research interests include reconfigurable architecture and machine learning.