PuDianNao: A Polyvalent Machine Learning Accelerator

Daofu Liu
SKLCA, ICT, China

Tianshi Chen *
SKLCA, ICT, China

Shaoli Liu
SKLCA, ICT, China

Jinhong Zhou
USTC, China

Shengyuan Zhou
SKLCA, ICT, China

Olivier Teman
Inria, France

Xiaobing Feng
SKLCA, ICT, China

Xuehai Zhou
USTC, China

Yunji Chen
SKLCA, CAS Center for Excellence in Brain Science, ICT, China

Abstract

Machine Learning (ML) techniques are pervasive tools in various emerging commercial applications, but have to be accommodated by powerful computer systems to process very large data. Although general-purpose CPUs and GPUs have provided straightforward solutions, their energy-efficiencies are limited due to their excessive supports for flexibility. Hardware accelerators may achieve better energy-efficiencies, but each accelerator often accommodates only a single ML technique (family). According to the famous No-Free-Lunch theorem in the ML domain, however, an ML technique performs well on a dataset may perform poorly on another dataset, which implies that such accelerator may sometimes lead to poor learning accuracy. Even if regardless of the learning accuracy, such accelerator can still become inapplicable simply because the concrete ML task is altered, or the user chooses another ML technique.

In this study, we present an ML accelerator called PuDianNao, which accommodates seven representative ML techniques, including k-means, k-nearest neighbors, naive bayes, support vector machine, linear regression, classification tree, and deep neural network. Benefited from our thorough analysis on computational primitives and locality properties of different ML techniques, PuDianNao can perform up to 1056 GOP/s (e.g., additions and multiplications) in an area of 3.51 mm^2, and consumes 596 mW only. Compared with the NVIDIA K20M GPU (28nm process), PuDianNao (65nm process) is 1.20x faster, and can reduce the energy by 128.41x.

1. Introduction

In the era of data explosion, Machine Learning (ML) techniques have become pervasive tools in emerging large-scale commercial applications such as social network, recommendation system, computational advertising, and image recognition. Facebook generates over 10 Petabyte (PB) log data per month [6]. Taobao.com, the largest online retailer in China, generates tens of Terabyte (TB) data every day [6]. The increasing amount of data poses great challenges to ML techniques, as well as computer systems accommodating those techniques.

The most straightforward way to accelerate large-scale ML is to design more powerful general-purpose CPUs and GPUs. However, such processors must consume a large fraction of transistors to flexibly support diverse application domains, thus can often be inefficient for specific workloads. In this context, there is a clear trend towards hardware accelerators that can execute specific workloads with very high energy-efficiency or/and performance. For ML techniques that have broad yet important applications in both cloud servers and mobile ends, of course, there have been some successful FPGA/ASIC accelerators, but each of which of-

* Tianshi Chen is the corresponding author. Address: State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China. Email: chentianshi@ict.ac.cn

ASPLOS '15, March 14–18, 2015, Istanbul, Turkey..

http://dx.doi.org/10.1145/2694344.2694358

ten targets at only a single ML technique or technique family. DianNao [7, 8] is one such example, which effectively accelerates representative neural network algorithms to benefit different ML scenarios (e.g., regression and classification).

While valid to support different ML scenarios at different scales, DianNao still has disadvantages. To be specific, when using DianNao to solve an ML problem, one has no choice but to use a neural network. Although recent studies have reported the large success of neural network (e.g., deep neural network), they were often achieved at great computational efforts, and involve too many parameters to tune. Even if regardless of the high computational complexity, neural network can still be worse than other ML techniques under certain scenarios. For example, in the classification of linearly-separable data, complex neural networks can easily become over-fitting, and perform worse than even a linear classifier. In application domains such as financial quantitative trading, linear regression is more widely-used than neural network due to the simplicity and interpretability of linear model [5]. The famous no-free-lunch theorem from the ML domain is a good summary of the above situation: any learning technique cannot perform universally better than another learning technique [39]. The insight here is that an ML accelerator should be able to support diverse ML techniques, in order to address needs under different scenarios.

However, unlike neural networks which share similar computational patterns, there is significant diversity among existing ML techniques, making it hard to design a pervasive ML accelerator. The diversity is two-fold. First, different ML techniques may differ a lot in their *computational primitives*. Here we take two classification algorithms, naive bayes and k-nearest neighbors (k-NN), as examples. The most time-consuming part of naive bayes is estimating the conditional probabilities, but that of k-NN is calculating distances between instances (each represented as a feature vector). Second, different ML techniques may differ a lot in their *locality properties*. For example, a large number of instances in k-NN may be frequently reused to classify unseen instances, while each feature (vector component) of an instance is not frequently reused in naive bayes. In this case, one must identify and factor in computational primitives and locality properties of representative ML techniques before deciding the overall accelerator architecture.

In this paper, we present an accelerator accommodating seven representative ML techniques, i.e., k-means, k-NN, naive bayes, support vector machine, linear regression, classification tree, and deep neural network. We conduct a thorough analysis of the ML techniques to extract critical computational primitives and locality optimizations that need to be supported by the accelerator. We present PuDianNao, an architecture design at TSMC 65nm process. On 13 critical phases of 7 representative ML techniques, the accelerator, clocked at 1GHz, is 1.20x faster and 128.41x more energy-efficient than the NVIDIA K20M GPU on average.

Our contributions are the following. First, we thoroughly analyze several representative ML techniques to extract their key computational tasks and locality properties, which establishes a solid foundation for the design of ML accelerator. Second, we design novel functional units to cover common computational primitives of different ML techniques, as well as on-chip storage that comprehensively factors in locality properties of different ML techniques. Third, we present PuDianNao, an accelerator accommodating seven representative ML techniques and multiple ML scenarios (e.g., classification, regression and clustering).

The rest of this paper proceeds as the following. Section 2 conducts a thorough analysis on computational primitives and locality properties of seven ML techniques. Section 3 presents the accelerator architecture of PuDianNao. Section 4 introduces the control module and code format of PuDianNao. Section 5 and 6 provide the experimental methodology and experimental results, respectively. Section 7 presents the related work.

2. Analysis of Representative ML Techniques: A Computer Architecture Perspective

In the machine learning community, an ML technique is conventionally characterized by its mathematical model (e.g., linear or non-linear), its learning style (e.g., supervised or unsupervised), its training algorithm (e.g., maximum-a-posteriori or gradient descent) and so on. From a computer architecture perspective, however, an ML technique is mainly characterized by its computational primitives and locality property. In this section, we do not repeat too much the context of ML textbooks, but provide an architect's-eye analysis of seven ML techniques.

For each ML technique, our analysis consists of two parts. In the first part, we identify the most time-consuming step of the technique. In the second part, we analyze the locality property of the identified time-consuming step of the ML technique, and leave the analysis/optimization on the rest of the technique to later sections. This divide-and-conquer idea well reflects the spirit of our accelerator (see Section 3.1): it accommodates the most time-consuming operations of ML techniques with dedicated hardware computational structures as well as optimized on-chip buffers, but supports the rest operations by integrating lightweight general-purpose Arithmetic Logic Units (ALUs).

Based on our locality analysis, we also investigate how to reduce off-chip memory bandwidth requirements of different ML techniques, and empirically check potential off-chip memory bandwidth reductions with an in-house cache simulator, which has 32KB cache (clocked at 1GHz) which has enough banks to support a 256-bit SIMD engine. To focus on memory behaviors, we assume that the SIMD engine can calculate any function with three 256-bit inputs (e.g., $f(a, b, c)$) at one cycle. We provide a sufficiently large num-

```
//Na is the number of testing instances
//Nb is the number of reference instances
for (i = 0; i < Na; i++){
  for(j = 0; j < Nb; j ++){
    Dis[i,j] = dis(t(i),r(j));
}  }
```

Figure 1: Original code of distance calculations.

ber of training/testing/reference instances (each is a 32x32-bit floating-point vector) for each ML technique when evaluating the off-chip memory bandwidth requirement.

2.1 k-Nearest Neighbors

k-Nearest Neighbors (k-NN) [2] is a simple yet widely-used learning algorithm that directly uses the k nearest neighbors of a testing instance to determine the label (classification or regression result) of the instance. For each testing instance, the k-NN has two major steps. First, it computes distances between the testing instance and reference instances. Second, it finds the k nearest reference instances to the testing instance, and assigns the testing instance the most frequent label among labels of the k nearest reference instances (classification), or the average label in the k nearest reference instances (regression). The most time-consuming operations in k-NN are calculating distances between instances. On UCI Gas datasets, distance calculations averagely account for 84.44% the computation time on an Intel Xeon E5-4620 CPU.

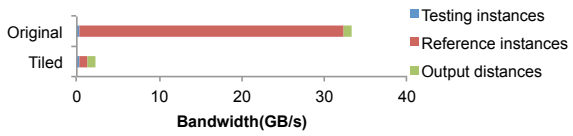

Figure 2: Memory bandwidth requirements for distance calculations of k-NN.

Figure 1 presents the original code of distance calculations. We observe that a reference instance can be reused after $N_b - 1$ distance calculations (N_b is the number of testing instances), while a testing instance can always be reused in its own loop. When N_b is very large, the reuse distance of a reference instance is also very large, and the chip cannot simultaneously store all reference instances, leading to frequent off-chip memory accesses that re-fetch the same reference instances for different testing instances. To exploit the locality of reference instances, we tile loops of both testing and reference instances, and define each *tiling block* to be distance calculations between T_i testing instances and T_j reference instances (see Figure 3). Tiling significantly reduces the reuse distance of reference instances, and can significantly reduce off-chip memory transfers. In our experiment comparing the off-chip memory bandwidth requirements before and after tiling, we set $T_i = T_j = 32$, i.e., $T_i + T_j = 64$ instances (each is a 32x32-bit floating-point vector) in a tiling block (64x32x4 Byte = 8KB) can be simultaneously stored in the cache (32KB). We observe that tiling reduces the off-chip memory bandwidth requirement of distance calculations by 93.9% (see Figure 2).

```
//Ti is the tiled testing block size
//Tj is the tiled reference block size
for (i = 0; i < Na/Ti; i++){
  for(j = 0; j < Nb/Tj; j ++){
    //tiled block
    for (ii = i*Ti; ii < (i+1)*Ti; ii++){
      for(jj = j*Tj; jj < (j+1)*Tj; jj++){
        Dis[ii,jj] = dis(t(ii),r(jj));
} } } }
```

Figure 3: Tiled code of distance calculations.

2.2 k-Means

k-Means [16] is an unsupervised ML technique which partitions N instances into k clusters. k-Means starts with k random cluster centroids, and iteratively performs two steps. First, it assigns each instance to the cluster whose current centroid (mean) is the nearest. Second, it computes the new mean of each cluster with the current members of the cluster, which then becomes the new centroid of the cluster. The most time-consuming operations in k-Means are distance calculations. On UCI Gas datasets, distance calculations account for 89.83% the computation time of k-Means on an Intel Xeon E5-4620 CPU.

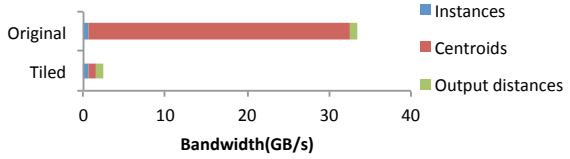

Figure 4: Memory bandwidth requirements for distance calculations of k-Means ($k = 64$).

In locality optimization, distance calculations in k-Means can be treated in a way (i.e., tiling) similar to what is done to k-NN. The minor difference here is that reference instances and testing instances in the locality analysis of k-NN are respectively replaced with cluster centroids (means), and instances to be clustered. We follow the same experimental setting of tiling in k-NN (e.g., instance length and tiling block size), and observe that tiling both cluster centroids and instances can reduce the off-chip memory bandwidth requirement of distance calculations by 92.5% (see Figure 4).

2.3 Deep Neural Network

Multi-Layer Perceptron (MLP) [12] is a classical artificial neural network that can model the non-linear relationship between inputs and outputs. Over the past decade, an emerging type of MLP called Deep Neural Network (DNN) has attracted broad interests of both the machine learning community and industry [10, 13, 20].

DNN is known to have a deep structure consisting of many hidden layers, and neurons in adjacent layers are often fully connected. A DNN has three computation modes, feedforward computation which computes the network output for each given input under the current network setting, pre-training which locally tune the synapses (connection weights) between each pair of adjacent layers, and global training which globally tune synapses with the Back Propagation (BP) algorithm [35].

We first analyze the feedforward computation, which computes neuron values of the $(g+1)$-th layer with neuron values of the g-th layer ($g \geq 1$). Without losing any generality, assume that we already know the neuron values of the g-th layer, a layer having N_a neurons $(x_1, \ldots, x_{N_a})$. We are going to compute the neuron values of the $(g+1)$-th layer, a layer having N_b neurons $(y_1, \ldots, y_{N_b})$. The value of the i-th neuron in the $(g+1)$-th layer, y_i, is computed by $y_j = f\left(\sum_{i=1}^{N_a} w_{ij}x_i + s_j\right)$, where f represents the activation function (e.g., sigmoid and tanh), w_{ij} is the synapse between neurons x_i and y_j, and s_j is the bias. The feedforward computation, which computes all N_b neurons in the same layer, can be written in a matrix form $Y = X \otimes W$, where

$$W = \begin{bmatrix} s_1 & s_2 & \cdots & s_{N_b} \\ w_{11} & w_{12} & \cdots & w_{1N_b} \\ w_{21} & w_{22} & \cdots & w_{2N_b} \\ \vdots & \vdots & \ddots & \vdots \\ w_{N_a1} & w_{N_a2} & \cdots & w_{N_aN_b} \end{bmatrix} \quad (1)$$

$X = (1, x_1, \ldots, x_{N_a})$, $Y = (y_1, \ldots, y_{N_b})$, and the operator $\otimes$ is similar to the conventional matrix/vector multiplication except that the dot product performed between each row vector of the left multiplier and each column vector of the right multiplier, say, $p \cdot q$ is now replaced with $f(p \cdot q)$. Here f is the activation function of neurons.

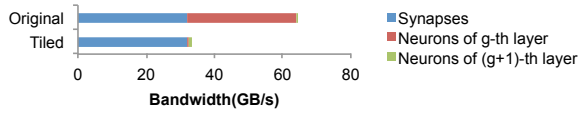

Figure 5: Memory bandwidth requirements for DNN feedforward computation ($N_a = 16384$).

It can be observed that neurons of the g-th layer ($x[i]$ in Figure 5) will be used for N_b times to calculate neurons in the next layer ($y[j]$ in Figure 5), while each synapse ($w[i,j]$ in Figure 6) is only be used once. In the original code, however, $x[i]$ can be reused after a total of $2N_a$ floating-point operations (multiplications and additions). When the number of neurons in the vector X (i.e., N_a) is very large, the reuse distance is also very large. In the meantime, the cache of a chip may not be sufficiently large to simultaneously store values of all N_a neurons, and have to re-fetch them when reusing. In order to reduce memory transfers, we tile the loop with respect to neurons in the vector X (see Figure 7). We also empirically evaluate the impact of tiling on feedforward computations, where we set $N_a = 16384$ (16384 neurons need a total of 16384×4Byte=64KB). We observe that tiling reduces the memory bandwidth requirement of feedforward computation by 46.7%.

```
//x is the input neurons, y is output
    neurous
//w is the weights, w[0,i] = s[i]
y(all) = 0; //initialize all neurons
for(i = 0; i < Nb; i++){
  for(j = 0; j < (Na+1); j ++){
    y[i] += w[j,i]*x[j];
    if(j == Na)
       y[i] = f(y[i]);
}  }
```

Figure 6: Original code of DNN feedforward computation.

```
//T is the tiled block size
y(all) = 0; //initialize all output
    neurons
for (j = 0; j < (Na+1)/T; j ++){
    //tiled block
    for(i = 0; i < Nb; i++){
       for(jj = j*T; jj < T; jj ++){
         y[i] += w[jj,i]*x[jj];
         if(jj == Na)
            y[i] = f(y[i]);
}  }  }
```

Figure 7: Tiled code of DNN feedforward computation.

Pre-training locally tunes synapses (weights) between each pair of adjacent layers, such synapses then serves as the initial synapses of global training. Pre-training can be done by training Restricted Boltzmann Machines (RBMs) [21], in which the most time-consuming steps are iterative feedforward computation and backforward computation (which includes similar operations to feedforward computation)[1]

[1] Given two adjacent layers A, B and synapses between them, a feedforward pass computes neuron values of layer B with neuron values of layer A and the synapses, while a backforward pass computes neuron values of layer A with neuron values of layer B and the same synapses. From a computer architecture perspective, they are the same.

invoked by Gibbs sampling. Global training uses the Back Propagation (BP) algorithm [35] to globally tune synapses, and the most-time consuming step is still analogous to feedforward/backforward computation. Therefore, the locality optimization done for feedforward computation also provides hints to pre-training and global training. We do not repeat details for the sake of brevity.

2.4 Linear Regression

Linear Regression (LR) [14] is a supervised learning technique which models the relationship between a scalar response variable y and a d-dimensional vector variable $x = (x_1, x_2 x_d)^T$ with a linear function $y = \sum_{i=0}^{d} \theta_i x_i$ (where x_0 always equals to 1). LR consists of two phases, the training phase which establishes the linear function from data, and the prediction phase which uses the existing linear model to predict the response (label) of each testing instance.

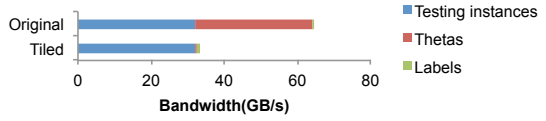

Figure 8: Memory bandwidth requirements for prediction phase of LR ($N_a = 16384$).

We starts with the prediction phase as it is simpler and more fundamental. At this phase, predicted responses of n different testing instances can be computed as

$$Y = \theta X, \tag{2}$$

where

$$X = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ x_1^{(1)} & x_1^{(2)} & x_1^{(3)} & \cdots & x_1^{(n)} \\ x_2^{(1)} & x_2^{(2)} & x_2^{(3)} & \cdots & x_2^{(n)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_d^{(1)} & x_d^{(2)} & x_d^{(3)} & \cdots & x_d^{(n)} \end{bmatrix}, \tag{3}$$

$\theta = (\theta_0, \theta_1, \ldots, \theta_d)$, and $Y = (y^{(1)}, \ldots, y^{(n)})$ are predicted responses of testing instances $x^{(1)}, \ldots, x^{(n)}$, respectively. This can be viewed as a multiplication between a vector and a matrix. The locality analysis is similar to that for feedforward computations in DNN (see Section 2.3). Briefly, the coefficients θ need to be reused, and there is no reuse of data in X. When the number of coefficients in a linear model is very large, we need to tile θ to reduce the reuse distance of those coefficients. Our empirical study on a setting of $d = 16384$ reveals that tiling reduces the memory bandwidth requirement of LR prediction phase by 46.7%.

At the training phase, the aim is to find appropriate coefficients $\theta = (\theta_0, \ldots, \theta_d)$ minimizing the Mean Square Error (MSE) over m training instances $\{x^{(1)}, y^{(1)}\}, \ldots, \{x^{(m)}, y^{(m)}\}$, where $y^{(i)}$ is observed response (label) of training instance $x^{(i)}$. Gradient descent is a common solution to this training task [41]. Briefly, gradient descent starts with an initial values of $\theta = (\theta_0, \ldots, \theta_d)$, and iteratively updates θ along the negative gradient direction. In gradient descent, the most-time consuming operations are calculating $\theta x^{(i)}$ $(i = 1, \ldots, m)$ with the latest θ. Such operations can also be written as Eq. (2). Therefore, the locality optimization for the prediction phase is also applicable here.

2.5 Support Vector Machine

Support Vector Machine (SVM) is a supervised learning technique for classification and regression [11]. The central idea of SVM is finding a hyperplane in a high-dimensional space, such that the distance between the hyperplane and the nearest training instance (i.e., geometric margin) is the largest. Like other supervised learning techniques, SVM has two phases, the training phase which constructed a model from the training data, and the prediction phase which use the trained model to predict the label of each testing instance. The SVM model, which predicts the label (y) of input instance x, has the form $y = \sum_{i=1}^{N} \alpha_i y^{(i)} k(x, x^{(i)}) + b$, where N is the number of training instances, $x^{(i)}$ and $y^{(i)}$ are the i-th training instance and its label respectively, α_i is a constant coefficient, $k(\cdot, \cdot)$ is a kernel function (e.g., radial basis function, tanh function) serves as the surrogate of computing the dot product between two instances at a high-dimensional space, and b is a constant. When $x^{(i)}$ is not a support vector[2], the corresponding coefficient α_i equals to 0.

At the SVM training phase, the goal is finding appropriate coefficients α_i that maximize the geometric margin, and a common training algorithm is Sequential Minimal Optimization (SMO) [32]. The most time-consuming step in SMO is to compute the $N \times N$ kernel matrix K (N is the total number of training instances), a symmetric matrix recording kernel function values of all possible pairs of training instances. Let k_{ij} $(i, j = 1, \ldots, N)$ be the entry at the i-th row and j-th column of matrix K, which is the kernel function value of training instances $x^{(i)}$ and $x^{(j)}$, i.e., $k_{ij} = k(x_i, x_j)$. Kernel matrix computation shares a similar locality property to distance calculations in k-NN, except that for each pair of instances, kernel matrix computation computes the value of kernel function instead of computing the distance. Therefore, we reuse the tiling mechanism designed for distance calculations. Tiled kernel matrix computation reduces the memory bandwidth requirement by 93.9%.

Assume that we have already constructed an SVM model involving N_a support vectors (see Footnote [2]). At the prediction phase, we use the constructed SVM model to predict labels of N_b testing instances. The most time-consuming computations at this phase are calculating kernel function values between each pair of support vector and testing in-

[2] A support vector is one of the training instances that are nearest to the decision hyperplane of SVM.

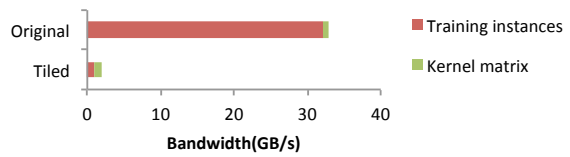

Figure 9: Memory bandwidth requirements for kernel matrix computation of SVM (d=32).

stance. The locality analysis is similar to that of distance calculations in k-NN, and the minor differences are that reference instances in k-NN are replaced with support vectors in the context of SVM, and for each pair of support vector and testing instance we calculate the kernel function value instead of their distance. Similarly, we can tile both support vectors and testing instances to reduce memory bandwidth requirement.

2.6 Naive Bayes

Naive Bayes (NB) is a probabilistic classifier based on Bayes' theorem and the Maximum-a Posteriori (MAP) paradigm [23]. It also relies on a strong assumption that instance features are independent with each other. NB has two phases: the training phase which estimates individual conditional probabilities from training instances, the prediction phase which classifies testing instances.

The goal of the training phase is to construct probability tables for estimating the posterior probability $p(C|F_1, \ldots, F_d)$, where C represents the label (class index), F_i $(i = 1, \ldots, d)$ represents the i-th feature. Here C and F_i are random variables, not specific values. By applying the Bayes' theorem and the independence assumption, $p(C|F_1, \ldots, F_d)$ can be written as $(1/Z)p(C)\prod_{i=1}^{d} p(F_i|C)$, where Z is a scaling factor that does not influence the results of NB. The central task of the training phase is to estimate conditional probabilities $p(F_i|C)$ $(i = 1, \ldots, d)$. Here we consider the discrete version of NB, thus those conditional probabilities can be obtained by performing frequency estimate on the training data. Assume that F_i $(i = 1, \ldots, d)$ can only take the value from $\{f_{ij}\}_{j=1}^{a}$, C can only take the value from $\{c_k\}_{k=1}^{b}$, then each conditional probability can be written as $p(F_i = f_{ij}|C = c_k)$, and there are a total of $d \times a \times b$ such items. NB maintains a *temporary counter* for each item to assist the frequency estimate. By streaming in features and label of training instances, NB completes all frequency estimates, and normalize the frequencies to get all conditional probabilities. In training an NB classifier, each dimension (feature value) of an instance is fetched to cache for a single time, but could be reused for multiple times when it is involved in, for example, multiple bitwise AND operations for comparing it with several candidate values that the feature can take (comparisons are due to conditional assignment/branching statements for choosing the right counter). Because each reuse of a feature value happens almost immediately after the last use, there is no need to tile instances as well as their feature values. During counting, however, temporary counters have to be frequently updated. When there are too many conditional probabilities to estimate, it is possible that the cache cannot store all temporary counters. To reduce memory transfers of temporary counters, one can pre-process training instances so that they are grouped according to their labels. Moreover, it is better to fetch from off-chip memory the corresponding features of different instances, rather than fetching each instance as a whole.

The prediction phase is relatively simpler. For each testing instance $x = (\hat{f}_1, \ldots, \hat{f}_d)$, NB calculates posterior probabilities $p(C = c_k|F_1 = \hat{f}_1, \ldots, F_d = \hat{f}_d)$ $(k = 1, \ldots, b)$, and assigns the class index that maximize the posterior probability as the label of instance X. The most time-consuming operations are multiplications of d numbers. Moreover, there is no significant locality here, as no one can predict the feature values of an unseen instance (which determine the data required by the follow-up multiplications).

2.7 Classification Tree

Classification Tree (CT) is a type of supervised learning techniques which construct tree-like classifiers from training instances. CT consists of two phases, the training phase which constructs a tree-like classifiers from training instances, and the prediction phase which predicts labels of testing instances.

At the training phase, CT starts with the root node representing the whole set of training instances, and recursively split non-leaf nodes into smaller nodes under the guide of a learning metric. Different CTs may use different learning metrics. For example, CART uses Gini impurity [3], ID3 uses information gain [33], and C4.5 uses information gain ratio [34]. However, the most time-consuming operations of all CTs are counting, which are indispensable steps for computing values of different learning metrics. There are often three types of counting tasks: counting the number of instances whose specific feature takes a specific value (for discrete feature space), counting the number of instances whose specific feature exceeds a threshold (for continuous feature space), and counting the number of instances having a specific label value. In each round of counting, the same feature value (vector component) of the same training instance (vector) will be frequently reused, because it is involved in, for example, multiple bitwise AND operations for comparing it with several candidate values that the feature can take (comparisons are due to conditional assignment/branching statements for choosing the right counter). Because each reuse of a feature value happens almost immediately after the last use, there is no need to tile instances as well as their feature values. In different rounds of counting, the same training instance could also be reused, but the reuse distance depends on data characteristics (instead of algorithm character-

istics), thus is not deterministic. Therefore, we cannot expect a predefined tiling strategy that can reduce the corresponding memory accesses.

At the prediction phase, the constructed CT classifier predicts the label of each testing instance. In this process, each testing instance needs to travel along the path from the root node to a leaf node, and the leaf node assigns the testing instance its label. When the size of the CT is very large, it is possible that the cache cannot store the whole tree, and there is an unacceptable cost if we reload the whole tree for each testing instance. To address this issue, we can follow the divide-and-conquer strategy and decompose the tree into sub-trees, each of which can be stored by cache. When a sub-tree is stored in the cache, it processes all testing instances that have not yet been labeled. This strategy can also be interpreted as tiling the tree, which avoids to frequently reload the whole tree.

2.8 Summary

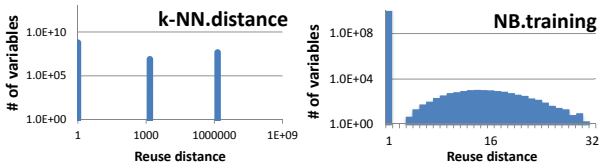

Figure 10: Average reuse distances of variables.

We provide a brief summary to this section. First, we identify some critical computational primitives of ML techniques, such as distance calculation, dot product, counting, as well as some non-linear functions, providing hints to implementing functional units of the accelerator. Second, and more importantly, we discover from our locality analysis that tiling can effectively exploit the data locality of k-NN, k-Means, DNN, LR, and SVM, but is not very helpful to NB and CT since they do not exhibit remarkable or predictable data locality. Figure 10 illustrates the average reuse distance (i.e., the average number of instructions between two consecutive accesses) of each variable (excluding loop variables) in k-NN (tiled distance calculations) and NB (training) on an x86 machine. As illustrated in Figure 10a, variables of distance calculations in k-NN naturally cluster into three classes in terms of their average reuse distances (similar behaviors can be observed from k-Means, DNN, LR, and SVM). This observation motivates us to use three separate on-chip buffers in the accelerator, each buffer stores variables having similar reuse distances (see Section 3.2 for details). Moreover, we observe from Figure 10b that variables in NB also cluster into two classes (similar behaviors can be observed from CT). The left class corresponds to feature and label values of training instances, whose reuse distances are always 1 (see Section 2.6 for more details). The right class corresponds to temporary counters, whose average reuse distances spread over a large interval. This is because the reuse of a temporary counter happens only when a specific feature of the current instance takes a specific value, which is a stochastic event decided by data characteristics instead of algorithm characteristics. For ML techniques like NB and CT, we need at least two on-chip buffers to fit two levels of reuse distances. When designing the accelerator, we carefully factor in observations made in this section. In follow-up sections, we will present the concrete accelerator architecture and show how the architecture accommodates different ML techniques.

3. Accelerator Architecture

In this section, we present the detailed architecture of PuDianNao, a hardware accelerator supporting seven representative ML techniques. As illustrated in Figure 11, the PuDianNao consists of several Functional Units (FUs), three data buffers (HotBuf, ColdBuf, and OutputBuf), an instruction buffer (InstBuf), a control module, and a DMA. In the rest of this section, we will elaborate the microarchitecture of each component.

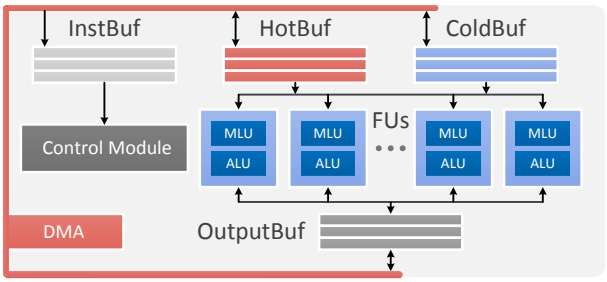

Figure 11: Accelerator architecture of PuDianNao.

3.1 Functional Units

Functional Units (FUs) are the basic execution units of the PuDianNao ML accelerator. Specifically, each FU consists of two parts, a Machine Learning functional Unit (MLU) and an Arithmetic Logic Unit (ALU).

3.1.1 Machine Learning Unit (MLU)

The MLU is designed to support several basic yet important computational primitives which are common in representative ML techniques, including dot product (LR, SVM, and DNN), distance calculations (k-NN and k-Means), counting (ID3 and NB), sorting (k-NN and k-Means), non-linear functions (e.g., sigmoid and tanh) and so on. As illustrated in Figure 12, the MLU is divided into 6 pipeline stages (Counter, Adder, Multiplier, Adder tree, Acc, and Misc).

In the **Counter** stage, each pair of inputs will be fed to a bitwise-AND unit or be compared by a comparer unit, and the value will then be added to an accumulator. After that, results of accumulators will be directly forwarded to

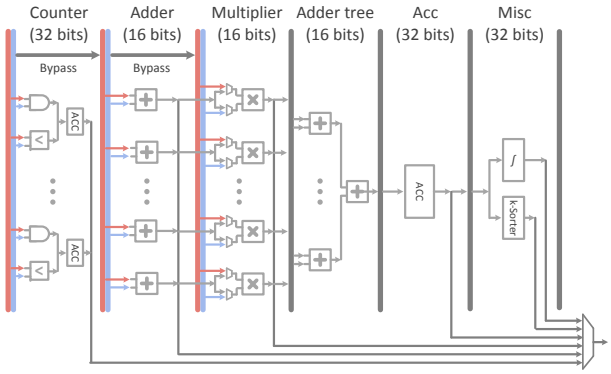

Figure 12: Implementation of MLU.

the output buffer rather than the next stage. This stage is used to accelerate counting operations in naive bayes and classification tree. When there is no need to execute counting operations, the counter stage will be bypassed.

The **Adder** stage is leveraged to operate vector addition, which is a very common operation in ML techniques. The adder stage may also be bypassed (e.g., in DNN), and results can be forwarded to either the next stage or the output port.

The **Multiplier** stage is leveraged to operate vector multiplication. Inputs of the multiplier stage can be the outputs of the previous stage (gray arrows in Figure 12) or data directly read from the input buffers (red and blue arrows in Figure 12). Results of the multiplier stage can be forwarded to either the next stage or the output port.

The **Adder tree** stage adds up results of all multipliers in the previous stage. The Adder tree stage and the Multiplier stage together support the dot product operation, an operation widely used in ML techniques (e.g. LR, SVM, and DNN). When the dimension of instance is larger than the size of adder tree, results of this stage are partial sums, which will be accumulated in the **Acc** stage. After obtaining the final result of dot product, the Acc stage will forward it to either the next stage or the output port.

The **Misc** stage integrates two modules, linear interpolation module and k-sorter module. The linear interpolation module is used to approximatively calculate non-linear functions involved in ML techniques (e.g. sigmoid and tanh in neural network). Different non-linear functions correspond to different interpolation tables. The k-sorter module is used to find the smallest k values from the outputs of Acc stage, which is a common operation in k-Means and k-NN. Results of linear interpolation module and k-sorter module will be forwarded to the output port respectively. Finally, the output port will select one out of the six collected results as the final output of the MLU module, as illustrated in Figure 12.

Furthermore, to reduce the area/power consumption of the PuDianNao accelerator, we implement 16-bit floating-point arithmetic units for stages Adder, Multiplier, and Adder tree. In the meantime, we still implement 32-bit floating-point units for the rest three stages (Counter, Acc, and Misc) to avoid potential overflow. We empirically evaluate the impact of the above hardware optimizations on the accuracies of k-NN, k-Means, SVM, LR, and MLP. We observe that it incurs negligible accuracy loss to these ML techniques (see Table 1), but significantly reduces the area of MLU. For example, we implemented the verilog of both 16-bit and 32-bit floating-point multipliers, placed/routed them, and found that the area of the 16-bit multiplier is only 20.07% the area of the 32-bit multiplier. Besides, here we do not evaluate naive bayes and classification tree, because they are not involved in stages Adder, Multiplier, and Adder tree, and will not be influenced by the 16-bit arithmetic units at all.

ML techniques	accuracy	
	all 16bits	32bits&16bits
SVM	37.7%	98.2%
k-NN	99.9%	100%
k-Means	93.9%	100.1%
LR	78.2%	99.0%
DNN	99.4%	100.1%

Table 1: Training accuracy of ML techniques under different sizes of arithmetic units (normalized to *all 32bits*).

3.1.2 Arithmetic Logic Unit (ALU)

In addition to computational primitives discussed above, in some ML techniques there are some other miscellaneous operations that are not supported by the MLU (e.g., division and conditional assignment). Although such operations are not very frequent in ML techniques, executing them on a host general-purpose core will still cause long-distance data movements as well as synchronization overheads. Therefore, we add a small Arithmetic Logic Unit (ALU) in each FU, which contains an adder, a multiplier, and a divider, as well as the converter of 32-bit float to 16-bit float and 16-bit float to 32-bit float. In addition, to support the $\log$ function required in training classification trees, we use ALU to compute approximations with the Taylor expansion of $\log(1-x)$. Our experiments of the ID3 classification tree on UCI datasets [1] suggest that computing $\log(\cdot)$ with the first 10 items of the Taylor series have been sufficient to remove the accuracy loss brought by approximations.

3.2 On-Chip Data Buffers

We reveal in Section 2 that tiling can effectively exploit the locality of many ML techniques, and we also observe that average reuse distances of variables in tiled ML techniques cluster into two or three classes (see Section 2.8). Motivated by this observation, we put three separate on-chip data buffers in the PuDianNao accelerator: HotBuf (8KB), ColdBuf (16KB) and OutputBuf (8KB). HotBuf stores the input data which have short reuse distance, and ColdBuf stores the input data with relative longer reuse distance. OutputBuf stores output data or temporary results.

In addition to the locality property, read width is another factor that motivates us to use multiple buffers. Here we take tiled distance calculations in k-Means as an example. Suppose that each time an MLU can only process f features (dimensions) of a testing instance and f features of a centroid. As the accelerator has a total of u MLUs, it can calculate partial square distances between one centroid and u testing instances at each cycle. Correspondingly, the MLUs need to read $f \times 16$ bits of centroids and $u \times f \times 16$ bits of testing instances at each cycle. In other words, the read width of centroids and testing instances are different. Therefore, we set two separate buffers (HotBuf and ColdBuf) to reduce the overhead brought by different read widths. HotBuf stores centroids, thus the read width is $f \times 16$ bits. HotBuff stores testing instances, thus the read width is $u \times f \times 16$ bits.

All three buffers are connected to the same DMA. Moreover, we use single-port SRAMs to construct HotBuf and ColdBuf, which can reduce the area and power consumption. As FUs may read (e.g. partial sum) or write the OutputBuf, we use dual-port SRAM to construct the OutputBuf.

CM	HotBuf				ColdBuf				OutputBuf								FU						
Inst Name	READ OP	READ ADDR	READ STRIDE	READ ITER	READ OP	READ ADDR	READ STRIDE	READ ITER	READ OP	WRITE OP	READ ADDR	WRITE ADDR	READ STRIDE	WRITE STRIDE	READ ITER	WRITE ITER	MLU-1 OP	MLU-2 OP	MLU-3 OP	MLU-4 OP	MLU-5 OP	MLU-6 OP	ALU OP

Table 2: Instruction format of PuDianNao.

CM	HotBuf				ColdBuf				OutputBuf								FU						
k-means	LOAD	0	16	128	LOAD	16384	256	16	NULL	STORE	NULL	1064960	NULL	16	NULL	16	NULL	SUB	SUB	MULT	ADD	NULL	SORT	NULL
k-means	READ	0	16	128	LOAD	20480	256	16	NULL	STORE	NULL	1065216	NULL	16	NULL	16	NULL	SUB	SUB	MULT	ADD	NULL	SORT	NULL
k-means	READ	0	16	128	LOAD	24576	256	16	NULL	STORE	NULL	1065472	NULL	16	NULL	16	NULL	SUB	SUB	MULT	ADD	NULL	SORT	NULL
.....																							
k-means	LOAD	2048	16	128	LOAD	16384	256	16	LOAD	STORE	1064960	1064960	16	16	16	16	NULL	SUB	MULT	ADD	NULL	SORT	NULL
.....																							

Table 3: An example of k-Means code ($f = 16$, $k = 1024$, $N = 65536$).

4. Control and Code Generator

The simplest way of controlling the accelerator is to hardwire all seven ML techniques. However, if the user wants to use another ML technique that is only slightly different from a hardwired ML technique, we might have to provide the user a new accelerator. To improve the flexibility of the accelerator, we use control instructions to support potential minor changes made on ML techniques. The accelerator only needs a single control module because all FUs will synchronously execute the same operations. The control module fetches instructions from the InstBuf, decodes the instructions, and sends operation signals to all FUs.

For each ML technique, the execution can be decomposed into several instructions. Table 2 presents the instruction format. More specifically, each instruction contains five slots: CM, HotBuf, ColdBuf, OutputBuf, and FU. This instruction format is customized for the PuDianNao architecture, thus a programmer needs to know implementation details of the accelerator. In order to facilitate programmers, we implement a code generator to generate instructions for different ML techniques.

We provide an example of generated k-Means code in Table 3. In this example, the number of features in each instance is $f = 16$, the number of centroids is $k = 1024$, and the number of testing instances is $N = 65536$. Centroids will be stored in the HotBuf (8KB), and testing instances will be stored in the ColdBuf (16KB). In order to hide the DMA memory access behind the computation, we leverage the HotBuf and ColdBuf in the ping-pong manner. Specifically, in the first instruction, the accelerator LOADs 128 centroids (4KB) and 256 testing instances (8KB) from the memory via the DMA, which occupy half of the HofBuf and ColdBuf, respectively. Then the accelerator calculates distances between the loaded centroids and testing instances. At the same time, another 256 testing instances are loaded to the other half of the ColdBuf, which will be leveraged by the second instruction. In the second instruction, the 128 centroids loaded in the first instruction will be reused, which are READ from the HotBuf. When distance calculations between the 128 centroids and all 65536 testing instances have been processed (after the 256-th instruction), a new block of 128 centroids will be loaded (in the 257th instruction). The process is repeated until distance calculations between all centroids and testing instances have been done.

5. Experimental Methodology

Evaluation. We use both Verilog and C simulator implementations of PuDianNao to measure the performance and power.

Verilog design. The design is synthesized by Synopsys Design Compiler with TSMC 65nm GP process in standard VT. The synthesized design is also placed and routed with the Synopsys ICC compiler. We then simulate and verify the design with the Synopsys VCS, and estimate the power consumption with Synopsys Prime-Tame PX based on the simulated Value Change Dump (VCD) file.

C simulator. Due to the slow simulation speed of the VCS, it is extremely time-consuming to run large-scale datasets on the verilog design of PuDianNao. To alleviate this problem, we implemented an in-house cycle-by-cycle C simulator of PuDianNao, and carefully calibrated it to the verilog design and TSMC library on small-scale datasets with $<0.3\%$ performance errors and $<7\%$ energy errors across all scenarios. The simulator allows a memory band-

ML technique	Dataset	Problem Size
k-NN	MNIST	60000 reference instances, 10000 testing instance 784 features, k=20
k-Means	MNIST	60000 instances, 784 features, k=10
DNN	MNIST	data size same with k-NN, L1=784,L2=L3=L4=L5=4096,L6=10
LR	MNIST	data size same with k-NN
SVM	MNIST	data size same with k-NN
NB	UCI Nursery	12960 instances, 8 features, 5 classes
Classification Tree(ID3)	UCI Covertype	522000 training instances, 59012 testing instances

Table 4: Benchmarks and dataset used in this paper.

width of up to 250GB/s, and is used to estimate the performance of PuDianNao on large-scale datasets.

Benchmarks. We evaluate PuDianNao and baseline processor on 7 ML techniques, including k-NN, k-Means, deep neural network, linear regression, support vector machine, naive bayes, and classification tree (ID3 [33]). We use MNIST [25] and UCI data [1] as benchmarks of ML techniques, see Table 4.

Baseline. GPUs have been widely used to support ML in industry due to their speed advantage over SIMD CPU. Hence, we take a modern GPU card (NVIDIA K20M, 3.52 TFlops peak, 5GB GDDR5, 208GB/s memory bandwidth, 28nm technology, CUDA SDK5.5) as the baseline. To validate the performance of the GPU baseline on ML applications, we compare it against a CPU with 256-bit SIMD (Intel Xeon E5-4620 Sandy Bridge-EP, 2.2GHz, 1TB memory, gcc compilation flag "-O3 -ftree-vectorize -march=native") over all benchmarks introduced above. As shown in Figure 13, our GPU baseline achieves an average speedup of 17.74x with respect to the SIMD CPU implementation. This result complies with two recent investigations which reported GPU to have 15x-49x [38] and 10x-60x [9] speedups over SIMD CPU for ML applications.

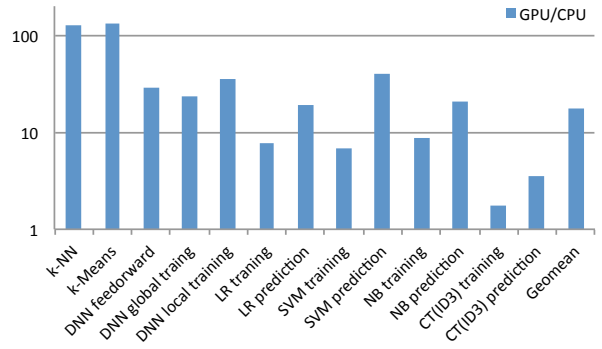

Figure 13: Performance comparison between GPU and SIMD CPU.

6. Experimental Results

In this section, we present characteristics of PuDianNao after layout, and quantitatively compare PuDianNao against the GPU baseline.

6.1 Characteristics after Layout

The current version of PuDianNao has 16 MLUs, each MLU can process 16 instance features (dimensions) at each cycle. Each MLU contains 16+16+15+1+1=49 adders, coming from the Counter stage, the Adder stage, the Adder tree stage, the Acc stage, and the Misc stage respectively. Each MLU contains 16+1=17 multipliers, coming from the Multiplier stage and the Misc stage, respectively. With 16 MLUs[3], PuDianNao can achieve a peak performance of $16 \times (49 + 17) \times 1 = 1056$ Gop/sec at 1GHz frequency, almost approaching the performance of a modern GPU.

However, the area and power consumption of PuDianNao are remarkably smaller than those of a modern GPU, by two orders of magnitude. According to the DC and ICC reports, the total area of PuDianNao is 3.51 mm^2, and the total power consumption is 596 mW. The critical path delay of PuDianNao is 0.99ns, suggesting that the accelerator can work at 1GHz frequency. The concrete area/power breakdowns are listed in Table 5. The on-chip buffers consume 62.64% and 31.37% of the total area and power. The combinational logic only consumes 21.97% area and 29.02% power. Among functional blocks, the most area-consuming part is ColdBuf (33.22%). All 16 FUs uses 19.38% area and 35.57% power. The concrete layout of PuDianNao is presented in Figure 14.

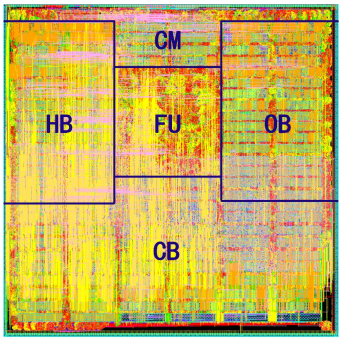

Figure 14: Layout of PuDianNao. CM, FU, HB, CB, and OB stand for Control Module, Functional Unit, HotBuf, ColdBuf, and OutputBuf, respectively.

6.2 Performance and Energy Consumption

We compare the performance of PuDianNao with GPU baseline across 13 phases of 7 ML techniques.[4] As shown in

[3] When calculating the peak performance, we voluntarily ignore contributions of ALUs, because ALU cannot work with MLU simultaneously.

[4] Many ML techniques have two phases each (training and prediction phases), but k-NN and k-Means only have one phase, and DNN has two different training phases, pre-training and global training.

Table 5: Characteristics of PuDianNao layout.

Component or Block	Area in μm^2	(%)	Power in mW	(%)	Critical path in ns
ACCELERATOR	3,513,437		596		0.99
Combinational	771,943	(21.97%)	173	(29.02%)	
On-chip buffers	2,201,138	(62.64%)	187	(31.37%)	
Registers	200,196	(14.23%)	86	(16.10%)	
Clock network	40,154	(1.14%)	143	(23.99%)	
Function Units	681,012	(19.38%)	117	(35.57%)	
ColdBuf	1,167,232	(33.22%)	78	(16.44%)	
HotBuf	578,829	(16.47%)	47	(9.56%)	
OutputBuf	586,361	(16.68%)	51	(10.23%)	
Control Module	481,737	(13.71%)	127	(21.30%)	
Other	18,266	(0.52%)	41	(0.06%)	

Figure 15, PuDianNao outperforms GPU on 6 phases out of the total 13 phases, and the average speedup of PuDianNao over the GPU is 1.20x. The maximal speedup of PuDianNao is achieved on *SVM prediction* (2.92x) , and an important reason is that PuDianNao provides dedicated linear interpolation functionality to efficiently compute kernel functions. The worst speedup of PuDianNao is achieved on *NB prediction* (0.37x). This phase needs frequent multiplications of different numbers (conditional probabilities) to obtain the overall posterior probabilities. PuDianNao does not have a large register file as the GPU, thus has to frequently move data between FUs and on-chip buffers, resulting in the observed performance loss. In contrast, the other phase of NB (*NB training*) does not have such operations, thus PuDianNao achieves a 2.22x speedup over the GPU on *NB training*.

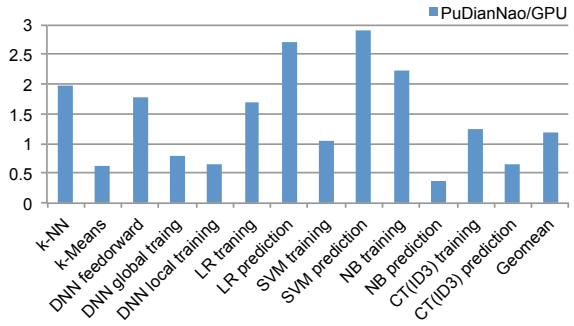

Figure 15: Performance speedup of PuDianNao over GPU.

We also compare energy consumptions of PuDianNao and the GPU baseline. As shown in Figure 16, PuDianNao averagely reduces the energy consumption of the GPU by 128.41x. This observation is not surprising, because PuDianNao is equipped with dedicated functional units and on-chip buffers optimized for the ML techniques. The largest energy reduction of PuDianNao is achieved on k-NN (262.20x), which is mainly because PuDianNao has efficiently supported sorting, a frequent and time-consuming operation (each finds top 20 instances out of 60000 reference instances in our experiment) in k-NN. In contrast, the GPU consumes remarkable energy on sorting with its general-purpose functional units. The smallest energy reduction of PuDianNao is achieved on *CT(ID3) prediction* (50.32x). That is because PuDianNao frequently reconfigures its DMA to support irregular memory accesses (e.g., linked list) for loading components of the ID3 classification tree. The above observations and discussions provide hints for further optimizations of the PuDianNao accelerator.

We do not factor in the energy/area of memory controller and off-chip memory of PuDianNao, but our energy/area comparison is still rather conservative: PuDianNao uses a 65nm process, while the GPU baseline uses a 28nm process. Although the energy/area reported for the GPU baseline does include the memory controller, this is far from compensating for the area ratio between 28nm and 65nm processes.

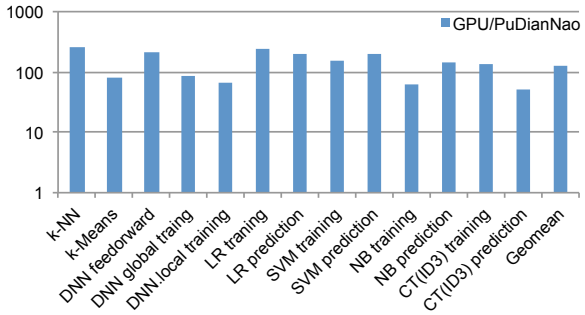

Figure 16: Energy reduction of PuDianNao over GPU.

7. Related Work

Machine learning has a broad application scope, but has to be accommodated by powerful computer systems to achieve high efficiency. Although it is straightforward to run ML techniques on general-purpose CPUs and GPUs (e.g., [18]), the efficiency is limited, because they spend too many efforts on flexibly supporting diverse application domains. For example, their functional units and memory hierarchies are not specifically designed to support ML applications.

There have been some successful ML accelerators that manage to implement dedicated hardware to accelerate machine learning. Yeh et al. designed a k-NN accelerator on FPGA [40]. Manolakos and Stamoulias designed two high-performance parallel array architectures for k-NN [29, 36]. There are also many dedicated accelerators for k-Means [17, 22, 30] or SVM [4, 31], due to their broad applications in industry. Recent debates on deep learning [24] even triggers the rebirth of *hardware neural network* [37], a hot topic in 1990s [19]. In this trend, a number of successful neural network accelerators have been proposed [7, 15, 26].

While remarkably increasing the energy-efficiency and performance compared with general-purpose processors, each of the above accelerators only accommodates a single technique or technique family (e.g.,neural network). When

the ML task shifts from, for example, classification to clustering, or the user favors another ML technique having better accuracy/efficiency, such an accelerator might easily become useless. Recently, Majumdar et al. proposed an accelerator called MAPLE which can accelerate matrix/vector operation and ranking used in five ML technique families (including neural network, SVM, k-means) [27, 28]. However, there are still many widely-used ML techniques (e.g., decision tree and naive bayes) whose major computational primitives are neither matrix nor vector operations (e.g., counting, linear interpolation), and such techniques cannot be supported by MAPLE. Furthermore, MAPLE does not excavate the locality properties of tiled ML algorithms, thus its memory bandwidth requirement might become the main bottleneck that prevents MAPLE from achieving high energy-efficiency/performance in large-scale ML applications common in industry.

Unlike previous investigations on ML accelerators which do not simultaneously address (a)*computational primitives* and (b)*locality properties* of (c)*diverse representative ML techniques*, we factor in all three dimensions during the design of PuDianNao accelerator. Although PuDianNao has not yet supported all representative ML techniques, the design methodology may shed some light on developing *general-purpose machine learning accelerator* in the future.

8. Conclusion

In this paper we present a machine learning accelerator called PuDianNao, which supports multiple ML scenarios (e.g., classification, regression and clustering) as well as multiple ML techniques (k-NN, k-Means, linear regression, SVM, DNN, naive bayes, classification tree). When executing the seven ML techniques, PuDianNao is 1.20x faster and 128.41x more energy-efficient than the NVIDIA K20M GPU on average. Compared with previous ML accelerators designed for narrow ranges of ML techniques, PuDianNao is more robust when the data characteristics change, or the application scenario is altered, because it provides users a basket of candidate techniques. This is somewhat similar to a common strategy adopted by Wallstreet traders: they usually invest to a basket of currencies to reduce the overall risk. Our future work on this topic includes extending PuDianNao to support some other mature ML techniques, as well as a tape-out of PuDianNao.

Acknowledgements

This work is partially supported by the NSF of China (under Grants 61100163, 61133004, 61222204, 61221062, 61303158, 61473275, 61432016, 61472396), the 973 Program of China (under Grants 2015CB358800, 2011CB302500), the Strategic Priority Research Program of the CAS (under Grant XDA06010403), the International Collaboration Key Program of the CAS (under Grant 171111KYSB20130002), a Google Faculty Research Award, the Intel Collaborative Research Institute for Computational Intelligence (ICRI-CI), the 10,000 talent program, and the 1,000 talent program.

References

[1] UC Irvine Machine Learning Repository. `http://archive.ics.uci.edu/ml/`. [Online; accessed 31-July-2014].

[2] Naomi S Altman. An introduction to kernel and nearest-neighbor nonparametric regression. *The American Statistician*, 46(3):175–185, 1992.

[3] Leo Breiman, Jerome Friedman, Richard Olshen, Charles Stone, D Steinberg, and P Colla. Cart: Classification and regression trees. *Wadsworth: Belmont, CA*, 156, 1983.

[4] Srihari Cadambi, Igor Durdanovic, Venkata Jakkula, Murugan Sankaradass, Eric Cosatto, Srimat Chakradhar, and Hans Peter Graf. A massively parallel fpga-based coprocessor for support vector machines. In *Field Programmable Custom Computing Machines, 2009. FCCM'09. 17th IEEE Symposium on*, pages 115–122. IEEE, 2009.

[5] Ernie Chan. *Algorithmic trading: winning strategies and their rationale*, volume 625. John Wiley & Sons, 2013.

[6] Min Chen, Shiwen Mao, Yin Zhang, and Victor CM Leung. *Big Data-Related Technologies, Challenges and Future Prospects*. Springer, 2014.

[7] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. Diannao: a small-footprint high-throughput accelerator for ubiquitous machine-learning. In *Proceedings of the 19th International Conference on Architectural support for programming languages and operating systems*, pages 269–284. ACM, 2014.

[8] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. Dadiannao: A machine-learning supercomputer. In *Proceedings of the 47th IEEE/ACM International Symposium on Microarchitecture (MICRO'14)*, pages 1–14. IEEE, 2014.

[9] Dan C Ciresan, Ueli Meier, Jonathan Masci, Luca Maria Gambardella, and Jürgen Schmidhuber. Flexible, high performance convolutional neural networks for image classification. In *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*, volume 22, page 1237, 2011.

[10] Ronan Collobert and Jason Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th international conference on Machine learning*, pages 160–167. ACM, 2008.

[11] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[12] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

[13] George E Dahl, Dong Yu, Li Deng, and Alex Acero. Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition. *IEEE Transactions on Audio, Speech, and Language Processing*, 20(1):30–42, 2012.

[14] Allen L Edwards. An introduction to linear regression and correlation. 1976.

[15] Clément Farabet, Berin Martini, Benoit Corda, Polina Akselrod, Eugenio Culurciello, and Yann LeCun. Neuflow: A runtime reconfigurable dataflow processor for vision. In *Computer Vision and Pattern Recognition Workshops (CVPRW), 2011 IEEE Computer Society Conference on*, pages 109–116. IEEE, 2011.

[16] Edward W Forgy. Cluster analysis of multivariate data: efficiency versus interpretability of classifications. *Biometrics*, 21:768–769, 1965.

[17] AC Frery, CC de Araujo, Haglay Alice, Jorge Cerqueira, Juliana A Loureiro, Manoel Eusebio de Lima, Mdas Oliveira, MM Horta, et al. Hyperspectral images clustering on reconfigurable hardware using the k-means algorithm. In *Integrated Circuits and Systems Design, 2003. SBCCI 2003. Proceedings. 16th Symposium on*, pages 99–104. IEEE, 2003.

[18] Vincent Garcia, Eric Debreuve, and Michel Barlaud. Fast k nearest neighbor search using gpu. In *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW'08. IEEE Computer Society Conference on*, pages 1–6. IEEE, 2008.

[19] Jan NH Heemskerk. Overview of neural hardware. *Neurocomputers for Brain-Style Processing. Design, Implementation and Application*, 1995.

[20] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdelrahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *Signal Processing Magazine, IEEE*, 29(6):82–97, 2012.

[21] Geoffrey E Hinton. Training products of experts by minimizing contrastive divergence. *Neural computation*, 14(8):1771–1800, 2002.

[22] Hanaa M Hussain, Khaled Benkrid, Huseyin Seker, and Ahmet T Erdogan. Fpga implementation of k-means algorithm for bioinformatics application: An accelerated approach to clustering microarray data. In *Adaptive Hardware and Systems (AHS), 2011 NASA/ESA Conference on*, pages 248–255. IEEE, 2011.

[23] Pat Langley, Wayne Iba, and Kevin Thompson. An analysis of bayesian classifiers. In *AAAI*, volume 90, pages 223–228. Citeseer, 1992.

[24] Quoc V Le. Building high-level features using large scale unsupervised learning. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 8595–8598. IEEE, 2013.

[25] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[26] Ahmed Al Maashri, Michael Debole, Matthew Cotter, Nandhini Chandramoorthy, Yang Xiao, Vijaykrishnan Narayanan, and Chaitali Chakrabarti. Accelerating neuromorphic vision algorithms for recognition. In *Proceedings of the 49th Annual Design Automation Conference*, pages 579–584. ACM, 2012.

[27] Abhinandan Majumdar, Srihari Cadambi, Michela Becchi, Srimat T Chakradhar, and Hans Peter Graf. A massively parallel, energy efficient programmable accelerator for learning and classification. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(1):6, 2012.

[28] Abhinandan Majumdar, Srihari Cadambi, and Srimat T Chakradhar. An energy-efficient heterogeneous system for embedded learning and classification. *Embedded Systems Letters, IEEE*, 3(1):42–45, 2011.

[29] Elias S Manolakos and Ioannis Stamoulias. Ip-cores design for the knn classifier. In *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, pages 4133–4136. IEEE, 2010.

[30] Tsutomu Maruyama. Real-time k-means clustering for color images on reconfigurable hardware. In *Pattern Recognition, 2006. ICPR 2006. 18th International Conference on*, volume 2, pages 816–819. IEEE, 2006.

[31] Markos Papadonikolakis and C Bouganis. A heterogeneous fpga architecture for support vector machine training. In *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on*, pages 211–214. IEEE, 2010.

[32] John C Platt, Nello Cristianini, and John Shawe-Taylor. Large margin dags for multiclass classification. In *nips*, volume 12, pages 547–553, 1999.

[33] J. Ross Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.

[34] J Ross Quinlan. Bagging, boosting, and c4. 5. In *AAAI/IAAI, Vol. 1*, pages 725–730, 1996.

[35] DE Rummelhart. Learning representations by back-propagating errors. *Nature*, 323(9):533–536, 1986.

[36] Ioannis Stamoulias and Elias S Manolakos. Parallel architectures for the knn classifier–design of soft ip cores and fpga implementations. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(2):22, 2013.

[37] Olivier Temam. The rebirth of neural networks. In *International Symposium on Computer Architecture*, 2010.

[38] George Teodoro, Rafael Sachetto, Olcay Sertel, Metin N Gurcan, W Meira, Umit Catalyurek, and Renato Ferreira. Coordinating the use of gpu and cpu for improving performance of compute intensive applications. In *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*, pages 1–10. IEEE, 2009.

[39] David H Wolpert. The lack of a priori distinctions between learning algorithms. *Neural computation*, 8(7):1341–1390, 1996.

[40] Yao-Jung Yeh, Hui-Ya Li, Wen-Jyi Hwang, and Chiung-Yao Fang. Fpga implementation of knn classifier based on wavelet transform and partial distance search. In *Image Analysis*, pages 512–521. Springer, 2007.

[41] Tong Zhang. Solving large scale linear prediction problems using stochastic gradient descent algorithms. In *Proceedings of the twenty-first international conference on Machine learning*, page 116. ACM, 2004.