LookNN: Neural Network with No Multiplication

Mohammad Samragh Razlighi[‡], Mohsen Imani[*], Farinaz Koushanfar[‡] and Tajana Rosing[*‡]
[‡]ECE Department, [*]CSE Department, UC San Diego, La Jolla, CA 92093, USA
{msamragh, moimani, fkoushanfar, tajana}@ucsd.edu

Abstract—**Neural networks are machine learning models that have been successfully used in many applications. Due to the high computational complexity of neural networks, deploying such models on embedded devices with severe power/resource constraints is troublesome. Neural networks are inherently approximate and can be simplified. We propose LookNN, a methodology to replace floating-point multiplications with look-up table search. First, we devise an algorithmic solution to adapt conventional neural networks to LookNN such that the model's accuracy is minimally affected. We provide experimental results and theoretical analysis demonstrating the applicability of the method. Next, we design enhanced general purpose processors for searching look-up tables: each processing element of our GPU has access to a small associative memory, enabling it to bypass redundant computations. Our evaluations on AMD Southern Island GPU architecture shows that LookNN results in $2.2\times$ energy saving and $2.5\times$ speedup running four different neural network applications with zero additive error. For the same four applications, if we tolerate an additive error of less than 0.2%, LookNN can achieve an average of $3\times$ energy improvement and $2.6\times$ speedup compared to the traditional GPU architecture.**

I. Introduction

Neural networks (NNs) are machine learning models that have shown promising accuracy in many tasks including but not limited to computer vision [1], voice recognition [2], [3], natural language processing [4], and health care [5]. Although NNs can outperform other machine learning models, they require enormous resources to be executed. Many applications require NNs to be executed on embedded devices. On the one hand, embedded devices are often constrained in terms of available processing resources and power budget. On the other hand, many applications such as pedestrian detection in automotive vehicles require NN execution to be both real-time and power efficient [6]. Parallel general purpose architectures such as GPUs are used to accelerate NNs, but their power consumption is not sustainable by embedded devices. Efficient ASIC accelerators can be designed for application-specific purposes but the design process is tedious. In addition, CMOS based arithmetic units bound the energy efficiency of ASIC designs.

NNs encompass multiple layers of neurons stacked in a hierarchical formation. Each layer takes the output of its preceding layer as a vector, then computes its own output in three steps: (i) The vector is multiplied by a 2-D matrix, (ii) a bias vector is added to the matrix-vector multiplication result, and (iii) a nonlinear function is applied. High dimensional floating-point matrix-vector multiplications are the most computationally complex operations in NN execution. In parallel processors, arithmetic operations are executed by slow and energy-hungry floating-point units (FPUs). To facilitate arithmetic operations of NNs, previous work utilizes quantized fixed-point parameters instead of floating-point parameters [7], [8]. Quantization is often accompanied by additive error, which can be reduced by adjusting the number of bits per numeral (bit-width). There are two major challenges associated with quantized NNs. First, training quantized parameters is problematic because numerical precision is indeed necessary in many applications [9]. Second, bit-width reconfiguration for accuracy control makes the design process tedious [10].

In this paper, we propose LookNN, a methodology to replace NN multiplications with look-up table search. For each neuron, LookNN stores all possible input combinations and the corresponding outputs in a look-up table. During execution, instead of utilizing the inefficient FPU, each neuron searches the look-up table to retrieve the pre-stored multiplication result. The proposed LookNN addresses the numerical precision requirement and the reconfigurable design requirement using a fixed bit-width (i.e. 32 bit floating point). LookNN can adjust the size of the look-up tables to trade-off between accuracy and efficiency.

Associative memory in a form of look-up table has been shown a great opportunity to improve the performance and energy efficiency of parallel processors [11], [12]. An associative memory can implement a look-up table that returns the search result in a single cycle. In this paper, we design associative memories to improve the execution time and power consumption of an AMD Southern Island GPU. Each processing element of the enhanced GPU has access to a small associative memory, enabling it to realize efficient look-up table search.

In summary, the main contributions of this paper are as follows:

- We propose LookNN, a methodology to replace NN multiplications with look-up table search.
- We provide theoretical analysis of LookNN's additive error. The analysis introduces design parameters to adjust the error. We design an algorithm that adapts conventional NNs to LookNN such that the additive error is minimized.
- We design associative memory blocks to reduce the power and execution time of FPUs in AMD Southern Island GPU. The enhanced GPU can realize efficient look-up table search. Our evaluations on four NN applications demonstrate an average of $2.5\times$ speedup and $2.2\times$ energy improvement with zero additive error.

II. Related Work

Using fixed-point quantized numerals in NNs is investigated in previous work [7]. Lin et al.[8] exploit trained binary parameters to avoid multiplication. Many applications require floating-point precision due to the fact that iterative training algorithms often update the parameters using gradients whose values are too small to sustain the additive quantization noise [9]. LookNN is similar to [8] in that it customizes the model during the training to reduce the additive error. LookNN is different since it enjoys floating-point precision rather than confining the parameters to binary numerals.

Model compression has been investigated in the literature. Han et al. [13] train sparse models with shared weights to

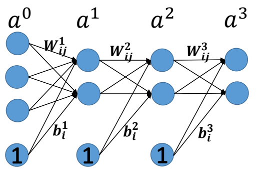


Fig. 1: Schematic depiction of a NN with two hidden layers. a^0 is the input vector, a^1 and a^2 are hidden layers, and a^3 is the output layer. $\{W^1, W^2, W^3\}$ denote the weights. Circles labeled with "1" denote virtual bias neurons.

compress the model. LookNN trains shared weights with a different approach. The shared weights in our model are subjective to a single neuron, making it possible to achieve higher accuracy with fewer shared weights per neuron. The compressed parameters of [13] can be used to realize ASIC/FPGA accelerators [14]. However, compression does not help with execution on general purpose processors, in which case the compressed parameters should be decompressed into the original parameters.

Dimensionality reduction is investigated for efficient execution of NNs [15]. Their method is orthogonal to LookNN; Our framework can receive pre-trained NNs and further reduce their power consummation and execution time.

This paper targets GPU implementation of lookNN using associative memory. Associative memory in a form of look-up table has been shown a great opportunity to improve the execution time and energy consumption of parallel processors [16], [12], [17]. Arnau et al. [12] utilize associative memories beside GPU floating-point units to enable error-free execution. Ternary content addressable memory (TCAM) is part of the associative memory that can search its contents in a single cycle. Recently, efficient TCAMs have been designed using high density and low leakage power non-volatile memories (NVMs) such as Spin Transfer Torque RAMs (STT-RAMs), ReRAM and Ferro electric RAMs (FeRAMs) [18], [19], [20]. Approximate associative memories using voltage overscaling and long time precharging are proposed to reduce FPU computations [17], [21]. However, in all previous work, computations rely on the FPUs and the computation efficiency is bounded by the processor's pipeline stage. To the best of our knowledge, LookNN is the first floating-point implementation on NNs that completely avoids using FPUs for multiplication.

III. Preliminaries

A. Neural Networks

Figure 1 depicts the schematic view of a NN. It encompasses multiple layers of neurons connected by feed-forward model parameters a.k.a. weights. The circles denote neurons, and the arrows represent model parameters. In a vectorized format, the NN computes each layer using the following equation:

$$a^l = f(W^l \times a^{l-1} + b^l) \tag{1}$$

where a^l and a^{l-1} denote layer l and its preceding layer respectively, W^l is the 2-D weight matrix connecting a^{l-1} and a^l, b^l is a bias vector, and f is an activation function such as "Tangent-hyperbolic". The most computationally intensive operations in NNs are dot-products of the form $< W_{i:}^T, a^{l-1} >$ where $W_{i:}^T$ denotes the transpose of row i in W^l.

During the training phase, the parameters of the model (i.e. weights and biases) are fine-tuned such that the model generates desired outputs. In the execution phase, the trained weights and biases are used to compute the output layer. Each neuron in the output layer represents a category; An output

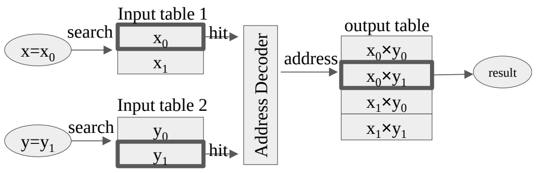


Fig. 2: Implementation of multiplier with look-up table. Each neuron of LookNN has one such look-up table. The input operands are ensured to come from finite sets, $\{x_1, x_2\}$ and $\{y_1, y_2\}$, which are pre-stored in the two input tables. Pairwise multiplications, $x_i \times y_j$, are pre-stored in the output table.

neuron's value is inferred as the probability that the input belongs to the category. The input is classified to the category with the highest probability. An accurate NN classifies most inputs to the correct category.

B. Multiplication with Associative Memory

Each neuron in LookNN maintains a look-up table illustrated in Figure 2, enabling the neuron to avoid using the FPU for multiplication. In order to make this implementation feasible, one should make sure that the operands (x, y) come from two finite sets. We ensure this property prior to LookNN execution. An associative memory is used to realized such tables as discussed in section IV-C.

IV. LookNN Framework

The global flow of our methodology is presented in Figure 3. The user defines the error tolerance, hardware constraints, the training data, and the baseline NN. Based on this information, the customization unit modifies the baseline NN and maps it into LookNN. The customization unit encompasses three major operations: weight clustering, error estimation, and weight retraining. Throughout this paper, we use the terms "clustered weights" and "shared weights" interchangeably. We use the term "customization" to denote error adjustment. Using our proposed recursive greedy algorithm described in

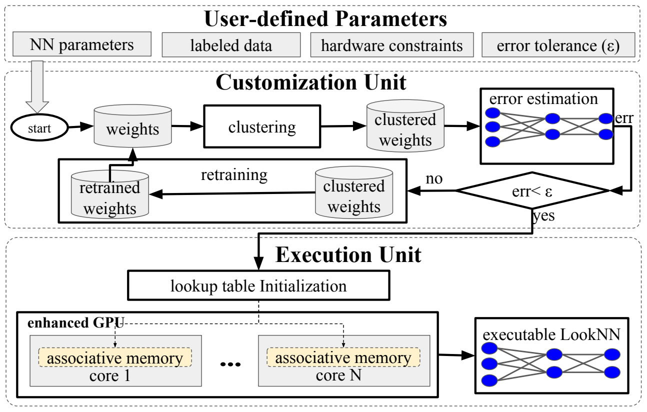


Fig. 3: Global flow of LookNN. The user assigns hardware constraints such as the look-up table size in the underlying hardware. The customization unit runs a greedy algorithm to adapt the NN to LookNN. The execution unit maps LookNN to the enhanced GPU and initiates the associative memories with proper values.

section IV-A, the customization unit trains a NN based on the hardware constraints (i.e. look-up table size) provided by the user. The algorithm has the following minimization objective:

$$\min_W (\Delta e) \quad s.t. \quad U(W) = N_{clusters}$$
$$\Delta e = e_{LookNN} - e_{baseline} \tag{2}$$

Where Δe is the additive error, e_{LookNN} and $e_{baseline}$ are the ratio of misclassified validation examples using LookNN and the baseline NN respectively, $U(W)$ is the number of distinct values in each row of the weight matrices, and $N_{clusters}$ is a pre-specified parameter provided by the user. $N_{clusters}$ directly translates to the power consumption and runtime in the execution phase.

After customization, the execution unit exploits the enhanced GPU to implement LookNN. Each GPU core has access to an associative memory whose energy consumption and runtime depend on the look-up tables' size. The associative memories are initialized with the multiplication operands and their pairwise multiplication results. NN customization and associative memory initialization are done once and their overhead is amortized across all future executions.

A. LookNN Customization Unit

Consider the look-up table in Figure 2. In LookNN, the first input table stores N_q rows representing the values that the preceding layer's neurons could possibly take. The second input table stores the neuron's incoming weights, $W^l_{i:}$, denoting a row of the 2-D matrix, W^l.

Algorithm 1 presents a pseudo code for the customization phase. It iterates a loop consisting of three major operations: weight clustering, error estimation, and weight retraining. Below we describe each operation in detail.

Algorithm 1 LookNN Customization Algorithm

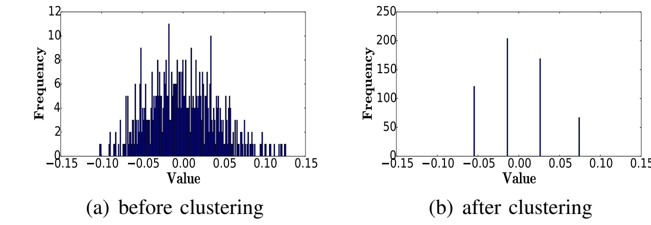
```
inputs: NN Parameters, $N_{clusters}$, Training Data, Error Tolerance ($\epsilon$)
outputs: LookNN Parameters, Validation Error
 1: $iteration \leftarrow 0$
 2: for $l = 1 \dots N_{layers}$ do
 3:     for $i = 1 \dots n_{l-1}$ do
 4:         $W^l_{i:} \leftarrow Kmeans(W^l_{i:}, N_{clusters})$
 5:     end for
 6: end for
 7: $e_{LookNN} \leftarrow err(W^{1 \dots N_{layers}}, b^{1 \dots N_{layers}}, x_{valid}, y_{valid})$
 8: if ($e_{LookNN} < \epsilon$ or $iteration == max\_iter$) then
 9:     return $W^{1 \dots N_{layers}}, b^{1 \dots N_{layers}}, e_{LookNN}$
10: end if
11: $retrain(W^{1 \dots N_{layers}}, b^{1 \dots N_{layers}}, x_{train}, y_{train})$
12: $iteration \leftarrow iteration + 1$
13: goto line 2.
```

weight clustering: Lines 2 through 6 of Algorithm 1 perform weight clustering. Each row of the matrix W^l is partitioned into $N_{clusters}$ clusters; The elements of the row are replaced by their closest centroids. The objective of clustering is to minimize the within cluster sum of squares (WCSS):

$$\min_{c_{i1},\dots,c_{iN_{clusters}}} \left(WCSS = \sum_{k=1}^{N_{clusters}} \sum_{W^l_{ij} \in c_{ik}} ||W^l_{ij} - c_{ik}||^2\right) \tag{3}$$

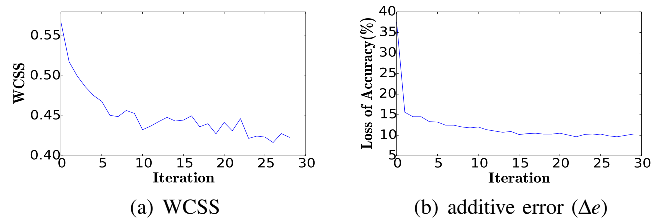
where $C = \{c_{i1}, c_{i2}, \dots, c_{iN_{clusters}}\}$ are the cluster centroids. We use K-means algorithm for clustering. Before clustering, the weights' values are scattered (Figure 4(a)) and the look-up table is large. After clustering, the number of rows in the look-up table is significantly decreased (Figure 4(b)).

weight retraining: Weight clustering is often accompanied by some degree of additive error, $\Delta e = e_{LookNN} - e_{baseline}$, to compensate for which we retrain the NN for a pre-specified number of epochs (line 11 of Algorithm 1). After retraining, the algorithm loops back to line 2 to cluster the retrained weights. Figure 5 depicts an example LookNN's additive error and the average WCSS of the weights. A retrained matrix is likely to exhibit less WCSS; Therefore, the error is reduced in subsequent iterations.



(a) before clustering (b) after clustering

Fig. 4: The distribution of a neuron's input weights. (a) The look-up table should maintain 561 rows. (b) The table should maintain $N_{clusters} = 4$ rows.



(a) WCSS (b) additive error (Δe)

Fig. 5: (a) The within cluster sum of squares (WCSS) is decreased in subsequent iterations. (b) The additive error is also decreased due to the reduction of the WCSS.

error estimation: The error estimation module computes the misclassification error, e_{LookNN}, over the validation data set (line 7 of Algorithm 1). This module simply computes the cross-validation error of LookNN with its clustered weights and quantized neurons. We use K-means to apply nonlinear quantization to the input layer, whereas hidden layers are quantized linearly.

B. LookNN Error Analysis

LookNN's additive error has two reasons: neuron quantization and weight clustering. We model the effect of both operations as additive Gaussian noise $N(\eta, \delta)$ with mean η and variance δ. Consider an $n \times m$ weight matrix W multiplied by an $m \times 1$ input layer a. We assume that clustering replaces each connection W_{ij} with $W_{ij} + N_{ij}(0, \delta_w)$ and quantization replaces each neuron a_j with $a_j + N(0, \delta_a)$. For simplicity, We also assume that the Gaussian noises are independent.

Consider a neuron computing a dot-product of the form $< W^T_{i:}, a^{l-1} >$. LookNN adds noise to the dot-product operands, resulting in the following noisy output:

$$\begin{aligned} z_{i-noisy} &= \sum_{j=1}^{m} (W_{ij} + N(0, \delta_w)) \times (a_j + N(0, \delta_a)) \\ &= z_{i-original} + N_i \end{aligned} \tag{4}$$

where $z_{i-noisy}$ is the noisy dot-product in LookNN, $z_{i-original}$ is the actual dot-product in the baseline NN and N_i is additive noise. The additive noise can be approximated as:

$$N_i \approx \sum_{j=1}^{m} W_{ij} \times N(0, \delta_a) + \sum_{j=1}^{m} a_j \times N(0, \delta_w) \tag{5}$$

N_i is a linear sum of independent Gaussian random variables with zero means; Therefore, it is a Gaussian random variable $N(0, \delta_i)$. The variance δ_i is obtained using equation 6:

$$\delta_i = \sum_{j=1}^{m} ||W_{ij}||^2 \times \delta_a + \sum_{j=1}^{m} ||a_j||^2 \times \delta_w \tag{6}$$

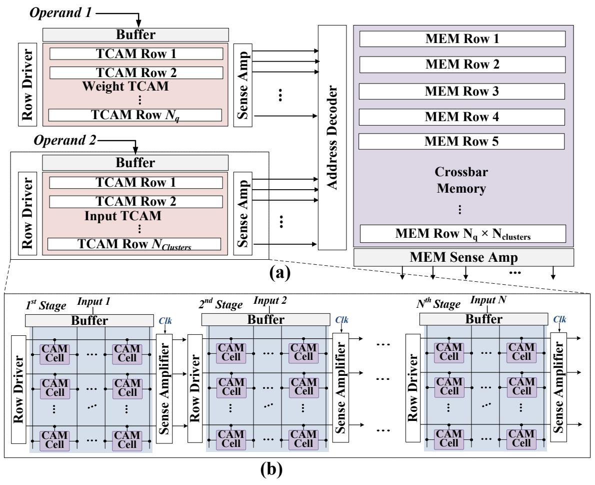


Fig. 6: (a) LookNN hardware for memory-based computation. N_q and $N_{clusters}$ determine the number of active rows in the TCAMs. (b) The structure of multistage TCAM.

The expected value of δ_i is:

$$E(\delta_i) = m \times (E(||W_{ij}||^2) \times \delta_a + E(||a_j||^2) \times \delta_w) \qquad (7)$$

δ_i denotes the variance of the quantization noise, δ_a, for the next hidden layer's neurons; Therefore, the noise over the next layers can be characterized in a similar manner. In fact, the noise propagates from the input layer through the output layer. The variance of the noise in the output layer is correlated with LookNN's error. High variance is interpreted as high probability of changing the actual NN's outputs. Equation 7 might not be perfectly exact for the general case, but it brings useful insights that we took into account for devising the customization process:

- The terms "δ_w" and "δ_a" suggest that minimizing the WCSS is effective for error reduction (see equation 3). The WCSS directly translates to "δ_w" of all layers and "δ_a" of the first layer.
- The expected value is proportional to "m", the number of input neurons in the preceding layer. Hence, matrices with higher number of elements per row require more shared values to reduce the WCSS.
- The term "$E(||W_{ij}||^2)$" demonstrates the importance of training regularized matrices. This can be achieved by applying Dropout [22] or adding a weight decay to the training minimization objective [23].

C. LookNN Execution Unit

Computing each neuron is assigned to one of the GPU streaming cores. Each core maintains an associative memory depicted in Figure 6. We use crossbar memristor, an access-free transistor memristive memory, to design both the TCAM and the crossbar memory. Prior to the execution, the first TCAM is initialized with N_q quantized values of the preceding layer. The second TCAM is initialized with $N_{clusters}$ shared weights. The crossbar memory holds $N_q N_{clusters}$ pairwise multiplication outputs. During execution, for a pair of input operands, the two TCAMs are searched in parallel, the address decoder generates the proper address, and the multiplication result is fetched from the crossbar memory. Below we discuss the memory blocks in detail.

TABLE I: Block size impact on energy consumption and search delay of LookNN ($N_q = 16$, $N_{clusters} = 16$, $bit-width = 32$)

	16-stage	**8-stage**	**4-stage**	**8-stage**	**1-stage**
Energy(fJ)	208	266	447	834	1307
Delay(ns)	0.80	0.58	0.28	0.17	0.11
EDP(J.s)$\times 10^{-24}$	167	156	128	148	150

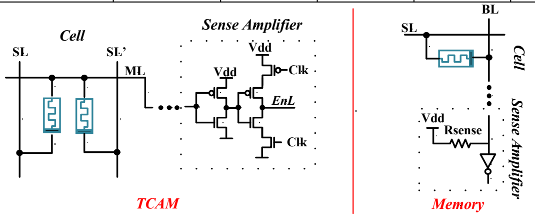


Fig. 7: TCAM and crossbar memory cells are implemented over CMOS FPU.

TCAM: Figure 6b illustrates the TCAM architecture. We design the TCAM using non-volatile memories. The values are stored on cells based on the NVM resistance state. Low and high resistances denote Logics 1 and 0 respectively. Before each search operation, the match lines (MLs) in all rows are pre-charged to Vdd voltage. During the search operation, a buffer distributes the input data among all rows. All MLs will discharge except the TCAM row matching the input data. The sense amplifier samples MLs at each clock cycle to identify the hit rows.

Mutli-Stage TCAM: In order to reduce TCAM's power consumption, we reduce its switching activity by splitting it into multiple stages. Figure 6b presents an N-stage TCAM. The first stage searches through 1/N of the data. The MLs of the subsequent stages are selectively pre-charged based on the hit rows of their preceding stages, resulting in significant energy saving. Dividing the TCAM into more stages reduces its energy consumption, but results in increased delay. Table I shows the trade-off for a table with $N_q = 16$ and $N_{clusters} = 16$. To choose the number of stages, we consider the energy-delay product (EDP) which is minimized using 4 stages.

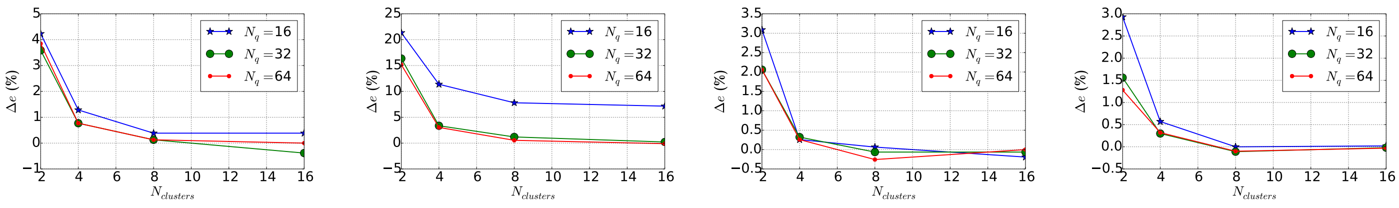
Crossbar Memory: Compared to the two TCAMs, the crossbar memory consumes lower energy. It occupies negligible area since we implement it in three dimensional (3D) architecture [24], [16]. The crossbar memory is implemented over the CMOS FPU, resulting in zero area overhead. Figure 7 shows the structure of TCAM and crossbar memory cells.

Associative Memory Configurations: LookNN can exchange accuracy for performance and energy efficiency. Increasing N_q and $N_{clusters}$ improves LookNN's accuracy at the cost of increase in energy consumption and execution time. The energy consumption of a look-up table search is characterized as:

$$E = E_T(N_q) + E_T(N_{clusters}) + E_C(N_q \times N_{clusters}) \qquad (8)$$

where E_T and E_C denote the energy consumption of TCAM and crossbar memory respectively. The execution time depends on the largest TCAM size, $max(N_q, N_{clusters})$, since the two TCAMs are searched in parallel. Table II shows LookNN search energy and delay in different associative memory configurations storing different numbers of patterns. Energy and delay are normalized to those of CMOS-based FPU multipliers. As the results show, all these configurations outperform FPU computation in both energy and delay. Our customization unit can adapt LookNN to any of these configurations.

Scalability: In large-scale NN, each streaming core is responsible for execution of multiple neurons. In such scenarios,



(a) Voice Recognition (b) Hyperspectral Imaging (c) Human Activity Recognition (d) MNIST

Fig. 8: Additive error of LookNN running four applications. The horizontal axis is the number of shared weights per neuron. The vertical axis represents $\Delta e = e_{LookNN} - e_{baseline}$. Each curve corresponds to a different neuron quantization.

TABLE II: LookNN normalized energy and delay in each configuration

	config1	*config2*	*config3*	*config4*	*config5*	*config6*
$(N_q, N_{clusters})$	(16,2)	(16,4)	(16,8)	(16,16)	(32,16)	(64,16)
Energy	0.03	0.04	0.05	0.07	0.12	0.26
Delay	0.12	0.13	0.15	0.18	0.23	0.34

we share each streaming core among multiple neurons of the same layer. The first TCAM does not need to be changed. The second TCAM should be extended with new weights. The crossbar memory should also be extended with new output values. For instance, *config1* of Table II can be extended to $(N_q, N_{clusters}) = (16, 16)$, allowing the associative memory to be shared among 8 neurons, each of which searches 2 rows of the second TCAM. The increase in the search energy is negligible since the number of searched rows is the same as the associative memory before being extended. The delay is neither affected since it depends on $max(N_q, N_{clusters})$.

V. Experimental Results

A. Experimental Setup

We integrate LookNN on the AMD Southern Island GPU, Radeon HD 7970 device including 2048 streaming cores. We perform circuit level simulations on HSPICE simulator using 45-nm TSMC technology. We use multi2sim, a cycle accurate CPU-GPU simulator for architecture simulation [25] and change the GPU kernel code to enable memory pre-loading and runtime simulation. We use Synopsys Design Compiler [26] to calculate the energy consumption of the 6-stage balanced FPUs in GPU architecture in 45-nm ASIC flow. NN applications are realized using OpenCL, an industry-standard programming model for heterogeneous computing. We use the Scikit-learn library [27] for Kmeans and Nearest Neighbour Search. Tensorflow [28] is used to realize the NN in the customization unit. We evaluate LookNN on four applications described below.

Voice Recognition: Many mobile applications require on-line processing of vocal data. We evaluate lookNN with the Isolet dataset [29] which consists of speech collected from 150 speakers. The goal of this task is to classify the vocal signal to one of the 26 English letters.

Hyperspectral Imaging: Hyperspectral imaging involves classification of different objects based on the reflectance spectra. The objective of this classification task is to recognize 9 different materials on the earth [30].

Human Activity Recognition: For this data set, the objective is to recognize human activity based on 3-axial linear acceleration and 3-axial angular velocity that have been captured at a constant rate of 50Hz [31].

TABLE III: Baseline NN and their error running four applications

Application	Network Topology $(\mathbf{l^0}, \mathbf{l^1}, \mathbf{l^2}, \mathbf{l^3})$	$\mathbf{e_{baseline}}$(%)
Voice Recognition	617, 500, 500, 26	4.4
Hyper-spectral Imaging	200, 500, 500, 9	6.6
Human Activity Recognition	561, 500, 500, 12	3.4
MNIST	784, 500, 500, 10	2.4

MNIST: MNIST is a popular machine learning data set including images of handwritten digits [32]. The objective is to classify an input picture to one of the ten digits $\{0 \dots 9\}$.

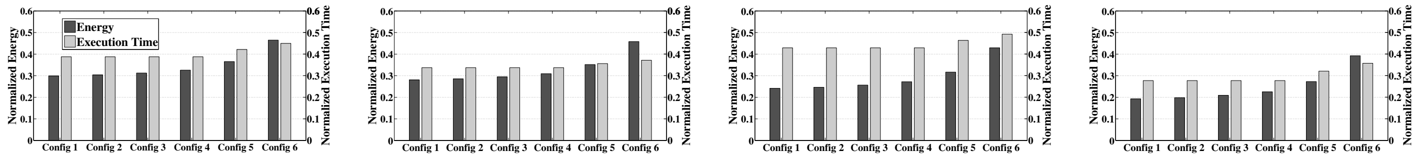
B. LookNN Evaluation

For each of the four data sets, we compare the baseline NN and its corresponding LookNN. The baseline utilizes the FPUs of the GPU wherase LookNN exploits the associative memory. Specifically, we compare them in terms of accuracy, running time and energy consumption. Stochastic gradient descent with momentum [33] is used for training. The momentum is set to 0.1, the learning rate is set to 0.001, and a batch size of 10 is used. Dropout [22] with drop rate of 0.5 is applied to hidden layers to avoid over-fitting. All data sets are normalized prior to the training, such that the features have 0 mean and standard deviation of 1. Table III presents the baseline NN Topologies and their error rates running four applications. The activation functions are set to "Rectified Linear Unit" clamped at 6. A "Softmax" function is applied to the output layer.

The additive error, $\Delta e = e_{LookNN} - E_{baseline}$, for different $(N_q, N_{clusters})$ configurations is depicted in Figure 8. It is clear that increasing N_q and $N_{clusters}$ results in reduced error. Note that for some configurations the error is negative (e.g. Voice recognition with $(N_q, N_{clusters}) = (32, 16)$), meaning that LookNN can achieve a lower error rate than the baseline; This happens due to the approximate nature of the baseline NN. For a fixed N_q, the error reduction exhibits diminishing return with respect to $N_{clusters}$. Therefore, both of the parameters $(N_q, N_{clusters})$ should be considered for error adjustment.

For each application, Table IV summarizes the additive error using selected LookNN configurations, each of which results in a different execution time and energy consumption. We report zero additive error for a negative Δe. A LookNN configuration can result in different error rates for different applications. This is due to the fact that some applications require precise numerical computations (e.g. Hyperspectral Imaging) while others can tolerate more numerical inaccuracy (e.g. MNIST).

Figure 9 depicts the energy consumption and execution time of LookNN normalized to those of the baseline NN. In both experiments, the overhead of data movement is accounted for. In order to get a zero Δe, our design requires to use associative memories at least in *config6* which results in



(a) Voice Recognition (b) Hyperspectral Imaging (c) Human Activity Recognition (d) MNIST

Fig. 9: Normalized energy consumption and execution time of LookNN in different configurations for four NN applications. The baseline NN is run on traditional GPU. LookNN is deployed on the enhanced GPU.

TABLE IV: Additive error in different LookNN configurations

	config1	*config2*	*config3*	*config4*	*config5*	*config6*
$(N_q, N_{clusters})$	(16,2)	(16,4)	(16,8)	(16,16)	(32,16)	(64,16)
Voice Recognition Δe	4.2%	1.3 %	0.4%	0.4%	0%	0%
Hyperspectral Imaging Δe	21%	11.4 %	7.8%	7.2%	0.2%	0%
Human Activity Recognition Δe	3.1%	0.25 %	0.2%	0%	0%	0%
MNIST Δe	3%	0.6 %	0%	0%	0%	0%

an average of 2.2× energy improvement and 2.5× speedup compared to the conventional AMD GPU architecture. In addition, our enhanced GPU achieves 3× energy improvement and 2.6× speedup if we tolerate an additive error of less than 0.2%. The trade-off is much more sensible if we solely consider the cost of multiplication, which is the main focus of this paper. For instance, compared to multiplication via FPU, LookNN achieves 33× energy improvement and 8.3× runtime improvement at *Config1*, while achieving 3.8× energy improvement and 3× runtime improvement at *Config6* (see TableII).

VI. Conclusion

We propose LookNN, a simplified NN that replaces multiplications with look-up table search, resulting in significant improvement in execution time and power consumption. Prior to converting NNs to LookNN, our customization unit can adjust NNs such that their accuracy is retained after converting to LookNN. The main advantage of LookNN over previous simplified models is that it enjoys floating-point parameters which is indeed necessary for many applications. LookNN can be deployed on either general purpose processors or FPGA/ASIC accelerators. Recently, associative memories have been used to enhance processors to bypass redundant computations. We employ one such enhanced GPU to evaluate LookNN. Our evaluations demonstrate an average of 2.2× energy improvement and 2.5× speedup with zero addtive error. LookNN can also be leveraged to exchange accuracy for efficiency; In our evaluations, LookNN achieves an average of 3× energy improvement and 2.6× speedup with an additive error rate of less than 0.2%.

VII. Acknowledgment

This work was supported in parts by NSF grant (1527034), ONR grant (N00014-11-1-0885), MURI grant (FA9550-14-1-0351), and Jacobs School of Engineering UCSD Powell Fellowship.

References

[1] Krizhevsky *et al.*, "Imagenet classification with deep convolutional neural networks," in *NIPS*, pp. 1097–1105, 2012.

[2] Hinton *et al.*, "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups," *Signal Processing Magazine, IEEE*, vol. 29, no. 6, pp. 82–97, 2012.

[3] Deng *et al.*, "Recent advances in deep learning for speech research at microsoft," in *ICASSP, 2013*, pp. 8604–8608, IEEE.

[4] Mikolov *et al.*, "Recurrent neural network based language model.," in *Interspeech*, vol. 2, p. 3, 2010.

[5] Srinivas *et al.*, "Applications of data mining techniques in healthcare and prediction of heart attacks," *IJCSE*, vol. 2, no. 02, pp. 250–255, 2010.

[6] Lane *et al.*, "Can deep learning revolutionize mobile sensing?," in *Proceedings of the 16th International Workshop on Mobile Computing Systems and Applications*, pp. 117–122, ACM, 2015.

[7] Lin *et al.*, "Fixed point quantization of deep convolutional networks," *arXiv preprint arXiv:1511.06393*, 2015.

[8] Lin *et al.*, "Neural networks with few multiplications," *arXiv preprint arXiv:1510.03009*, 2015.

[9] Lin *et al.*, "Overcoming challenges in fixed point training of deep convolutional networks," *arXiv preprint arXiv:1607.02241*, 2016.

[10] Reagen *et al.*, "Minerva: Enabling low-power, highly-accurate deep neural network accelerators," in *Proceedings of ISCA*, 2016.

[11] Imani *et al.*, "Acam: Approximate computing based on adaptive associative memory with online learning," in *ISLPED*, 2016.

[12] Arnau *et al.*, "Eliminating redundant fragment shader executions on a mobile gpu via hardware memoization," in *ISCA*, pp. 529–540, IEEE, 2014.

[13] Han *et al.*, "Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding," *CoRR, abs/1510.00149*, vol. 2, 2015.

[14] Han *et al.*, "Eie: efficient inference engine on compressed deep neural network," *arXiv preprint arXiv:1602.01528*, 2016.

[15] Rouhani *et al.*, "Delight: Adding energy dimension to deep neural networks," in *ISLPED*, pp. 112–117, ACM, 2016.

[16] Imani *et al.*, "Exploring hyperdimensional associative memory," in *HPCA*, IEEE, 2017.

[17] Imani *et al.*, "Resistive configurable associative memory for approximate computing," in *2016 DATE*, pp. 1327–1332, IEEE, 2016.

[18] Yin *et al.*, "Exploiting ferroelectric fets for low-power non-volatile logic-in-memory circuits," in *Proceedings of the 35th International Conference on Computer-Aided Design*, p. 121, ACM, 2016.

[19] Beigi *et al.*, "Tapas: Temperature-aware adaptive placement for 3d stacked hybrid caches," in *Proceedings of the Second International Symposium on Memory Systems*, pp. 415–426, ACM, 2016.

[20] Kim *et al.*, "Cause: critical application usage-aware memory system using non-volatile memory for mobile devices," in *ICCAD*, pp. 690–696, IEEE Press, 2015.

[21] Imani *et al.*, "Masc: Ultra-low energy multiple-access single-charge tcam for approximate computing," in *DATE*, pp. 373–378, IEEE, 2016.

[22] Srivastava *et al.*, "Dropout: a simple way to prevent neural networks from overfitting.," *Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[23] Moody *et al.*, "A simple weight decay can improve generalization," *NIPS*, vol. 4, pp. 950–957, 1995.

[24] Kim *et al.*, "A functional hybrid memristor crossbar-array/cmos system for data storage and neuromorphic applications," *Nano letters*, vol. 12, no. 1, pp. 389–395, 2011.

[25] Ubal *et al.*, "Multi2sim: a simulation framework for cpu-gpu computing," in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pp. 335–344, ACM, 2012.

[26] D. Compiler, R. User, and M. Guide, "Synopsys," *See: http://www.synopsys.com*.

[27] Pedregosa *et al.*, "Scikit-learn: Machine learning in python," *Journal of Machine Learning Research*, vol. 12, no. Oct, pp. 2825–2830, 2011.

[28] Abadi *et al.*, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *arXiv preprint arXiv:1603.04467*, 2016.

[29] "Uci machine learning repository." http://archive.ics.uci.edu/ml/datasets/ISOLET.

[30] "Hyperspectral remote sensing scenes." http://www.ehu.eus/ccwintco/index.php?title=Hyperspectral_Remote_Sensing_Scenes.

[31] "Uci machine learning repository." http://archive.ics.uci.edu/ml/datasets/Human+Activity+Recognition+Using+Smartphones.

[32] Y. LeCun, C. Cortes, and C. J. Burges, "The mnist database of handwritten digits," 1998.

[33] Sutskever *et al.*, "On the importance of initialization and momentum in deep learning.," *ICML (3)*, vol. 28, pp. 1139–1147, 2013.