2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture

Computation Reuse in DNNs by Exploiting Input Similarity

Marc Riera, Jose-Maria Arnau, Antonio González
Department of Computer Architecture
Universitat Politécnica de Catalunya (UPC), Barcelona, Spain
Email: {mriera, jarnau, antonio}@ac.upc.edu

***Abstract*—In recent years, Deep Neural Networks (DNNs) have achieved tremendous success for diverse problems such as classification and decision making. Efficient support for DNNs on CPUs, GPUs and accelerators has become a prolific area of research, resulting in a plethora of techniques for energy-efficient DNN inference. However, previous proposals focus on a single execution of a DNN.**

Popular applications, such as speech recognition or video classification, require multiple back-to-back executions of a DNN to process a sequence of inputs (e.g., audio frames, images). In this paper, we show that consecutive inputs exhibit a high degree of similarity, causing the inputs/outputs of the different layers to be extremely similar for successive frames of speech or images of a video.

Based on this observation, we propose a technique to reuse some results of the previous execution, instead of computing the entire DNN. Computations related to inputs with negligible changes can be avoided with minor impact on accuracy, saving a large percentage of computations and memory accesses.

We propose an implementation of our reuse-based inference scheme on top of a state-of-the-art DNN accelerator. Results show that, on average, more than 60% of the inputs of any neural network layer tested exhibit negligible changes with respect to the previous execution. Avoiding the memory accesses and computations for these inputs results in 63% energy savings on average.

***Keywords*-DNN; Computation Reuse; Input Similarity; Hardware Accelerator**

I. Introduction

Deep Neural Networks (DNN) represent a machine learning approach that delivers the most effective solution to a broad range of applications. DNN-based systems are ubiquitous and, thus, providing energy-efficient DNNs is a key feature for current and future computing devices. Previous hardware-accelerated DNN systems [1]–[4] provide efficient implementations of fully-connected and convolutional layers, optimized for an isolated execution of the DNN. Our work is motivated by the observation that many computations and memory accesses are redundant if we take into account successive executions of a DNN, especially for applications that process a temporal sequence of inputs (e.g., speech, video).

Figure 1 illustrates the case of speech recognition, where a DNN is executed multiple times to classify a sequence of audio frames in phonemes. Consecutive DNN executions have extremely similar inputs due to two main reasons. First, the length of these frames is in the order of several milliseconds. The speech signal is quasi stationary for such a short interval and, therefore, consecutive frames exhibit a high degree of similarity. Second, the DNN uses context information (neighbor frames) to classify each audio frame and, hence, successive executions operate on overlapping windows of frames. Note that other applications such as video processing exhibit similar behavior, as consecutive images in a video tend to be very similar.

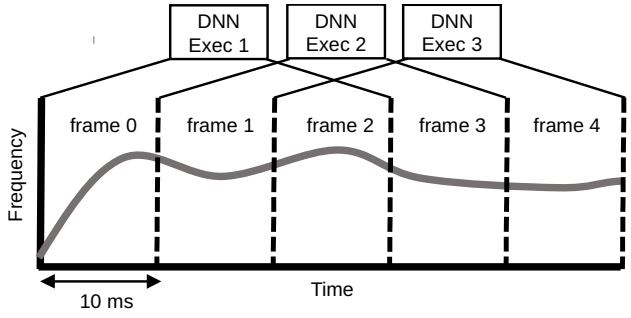

Fig. 1. In speech recognition, the audio signal is split in frames of 10 ms. The DNN is executed multiple times to classify the frames in phonemes. In this example, each DNN execution takes as input a window of three frames.

Despite the high degree of similarity, floating point computations are not exactly the same in two successive DNN executions in the vast majority of the cases. However, DNNs are known to be error tolerant [1]. We leverage this property to boost the potential of reuse across successive executions of the DNN. In particular, we found that linear quantization [4] of inputs is a very effective mechanism to increase the ability of our technique to exploit redundancy.

In this paper, we show that by applying linear quantization to the inputs of various DNNs for speech recognition [5]–[7], video classification [8] and self-driving cars [9], more than 60% of the inputs of fully-connected and convolutional layers have the same quantized value as in the previous DNN execution, whereas quantization has a minimal impact in accuracy. In addition, we also consider Recurrent Neural Networks (RNNs), a well-known alternative for sequence processing problems. More specifically, we employ an RNN for speech recognition [10] and show that, when applying linear quantization, more than 50% of the inputs in recurrent layers remain unmodified with respect to the prior execution, with negligible impact in accuracy.

2575-713X/18/$31.00 ©2018 IEEE
DOI 10.1109/ISCA.2018.00016

Based on the high degree of redundancy for the inputs of consecutive DNN executions, we propose a mechanism that computes the outputs of each DNN layer by reusing the buffered results for the previous execution. A simple example helps to illustrate our proposal. Let us consider a neuron of a fully-connected layer with three inputs. For the first execution of the DNN, the output z^1 is computed as follows: $z^1 = i_1^1 w_1 + i_2^1 w_2 + i_3^1 w_3 + b$, where i, w and b are the inputs, weights and bias respectively. In a similar way, the output of this neuron in the second DNN execution is $z^2 = i_1^2 w_1 + i_2^2 w_2 + i_3^2 w_3 + b$. However, if the first two inputs are the same in both executions, the output can be computed more efficiently as: $z^2 = z^1 + (i_3^2 - i_3^1) w_3$. In other words, we just need to subtract the old inputs that are different and add the new ones (multiplied by their respective weights). Note that in this case only one weight has to be fetched from memory instead of three, the bias is not required and only three floating-point computations are performed instead of six. Moreover, the subtraction of the two inputs can be reused for all the neurons in the same layer, so its cost is practically negligible and, in practice, for the above example the total cost approaches to just two floating point operations. Therefore, for DNNs that exhibit some degree of similarity in the inputs across multiple executions, it is more efficient to compute the current output in this way rather than evaluating again the entire DNN from scratch.

In this paper, we propose a hardware implementation of this reuse-based DNN inference mechanism. We extend the architecture of a state-of-the-art DNN accelerator to buffer the outputs of the different layers and reuse them for the next execution. The accelerator quantizes the inputs of each fully-connected, convolutional and recurrent layer, and then it computes the output by using only the inputs that have changed with respect to the previous execution, avoiding the corresponding memory accesses and computations for the inputs that remain unmodified.

The extra hardware required for our technique is modest, as the components already available for DNN computation can also be employed for most of the operations, such as quantization. In addition to small changes in the control unit, our technique only requires extra storage for the outputs of each layer. The extra memory represents a small increase in the on-chip storage of the accelerator. Our experimental results show that the overheads are minimal compared to the savings in memory fetches and computations, so the proposed scheme achieves a reduction in energy consumption of 63% on average for several DNNs.

To summarize, this paper focuses on energy-efficient, real-time DNN inference. Sequence to sequence learning is a broad area with numerous and important applications such as speech recognition, machine translation, video description or language modeling. According to numbers published by Google [11], at least 29% of Google's datacenter workloads are sequence processing. In comparison, only 5% of Google's workloads are CNNs for classification. Our computation reuse scheme can be applied to any sequence processing problem. Due to the small overheads, only a small degree of input similarity across DNN executions is required to achieve savings in computations and energy. The main contributions of this paper are the following:

- We analyze the degree of similarity in the inputs of all neurons for consecutive executions of a DNN, using several neural networks for speech recognition, video classification and self-driving cars. We show that, on average, more than 60% of the inputs of any fully-connected and convolutional layer have the same value in two consecutive executions, whereas more than 50% of the inputs of recurrent layers remain unmodified.
- We propose a novel DNN architecture that computes the output of fully-connected, convolutional and recurrent layers by exploiting the similarity in the inputs. This technique reduces the memory accesses and the computations by 66% on average.
- We implement a reuse-based inference technique on top of a state-of-the-art DNN accelerator. We show that our technique improves its performance by 3.5x and reduces its energy consumption by 63% on average.

The rest of the paper is organized as follows. Section II reviews the DNNs used in this paper. Section III presents the analysis of similarity in the inputs of a DNN. Section IV describes the hardware implementation of the reuse-based DNN inference technique. Section V presents the evaluation methodology and Section VI discusses the experimental results. Section VII reviews some related work and, finally, Section VIII sums up the main conclusions.

II. Deep Neural Networks

Proposals for solving sequence processing problems can be classified in three types of Deep Neural Networks (DNNs). In first place, a MultiLayer Perceptron (MLP) consists of multiple Fully-Connected (FC) layers in which every input is connected with every neuron. MLPs have been successfully used for a broad set of problems such as speech recognition [5] or machine translation [12] among other applications. In second place, a Convolutional Neural Network (CNN) combines both convolutional [13] and FC layers, and they have proved to be particularly efficient for image and video processing [8]. Finally, a Recurrent Neural Network (RNN) includes recurrent layers with feedback connections. RNNs can retain information of past inputs to improve the accuracy of future predictions.

Although RNNs have achieved tremendous success for some sequence processing problems [10], [14], MLPs and CNNs deliver state-of-the-art accuracy for applications such as acoustic scoring in speech recognition [5] or self-driving cars [9] respectively. To illustrate the broad applicability of our computation reuse technique, we evaluate our proposal on MLPs, CNNs and RNNs for a diverse set of applications. The next subsections provide more details on FC, convolutional and recurrent layers.

A. Fully-Connected Layers

The main performance and energy bottleneck in MLPs are the FC layers. In an FC layer, each neuron performs a dot

product operation between all the inputs of the layer and its corresponding weights. More specifically, the output of neuron j is computed according to the following equation:

$$out(j) = (\sum_{i=0}^{N} w_{ij} * in(i)) + b_j \quad (1)$$

where $in(i)$ represents the input vector, w_{ij} is the weight of neuron j for input i, and b_j represents the bias of neuron j. An FC layer with N inputs and M neurons contains $N \times M$ weights, as each neuron has its own weights, and requires $2 \times N \times M$ floating point operations. FC layers employed in real applications consists of thousands of neurons and they typically account for most of the computations and memory bandwidth usage in MLPs. FC layers also take most of the storage requirements and memory bandwidth usage in CNNs.

B. Convolutional Layers

Convolutional layers are commonly used for image processing, object recognition and video classification. A convolutional layer implements a set of filters to detect features in the input image. For video processing, 3D convolution across multiple frames provides higher accuracy than 2D convolution [8]. In case of 3D convolution, a filter is defined by $K_x \times K_y \times K_z$ coefficients or weights. Each convolutional layer applies multiple of these filters through the entire input, resulting in multiple output feature maps. Note that, unlike what happens with FC layers, the weights of the different filters are shared by all the neurons in the layer. The concrete formula for calculating an output neuron $out(x, y, z)^{f_o}$ at position (x, y, z) of output feature map f_o is:

$$out(x,y,z)^{f_o} = \sum_{f_i=1}^{N_{if}} \sum_{k_x=0}^{K_x} \sum_{k_y=0}^{K_y} \sum_{k_z=0}^{K_z} (w_{f_i,f_o}(k_x,k_y,k_z) * in(x+k_x, y+k_i, z+k_z)^{f_i}) \quad (2)$$

where $in(x, y, z)^{f_i}$ represents the input at position (x, y, z) in feature map f_i, and $w_{f_i,f_o}(k_x, k_y, k_z)$ is the synaptic weight at kernel position (k_x, k_y, k_z) in input feature map f_i for filter f_o. Convolutional layers take most of the computations performed in CNNs.

C. Recurrent Layers

Recurrent layers include loops or feedback connections, allowing information to persist from one execution of the network to the next. Long Short Term Memory (LSTM) cells [15] represent the most successful implementation due to its ability to capture long term dependencies, as they are able to keep useful information for future predictions over long periods of time. Figure 2 shows a bidirectional LSTM layer that includes two different LSTM cells executed in the forward and backward direction respectively. As it can be seen, the same LSTM cell is recurrently executed for each element x_t in the input sequence. Furthermore, the cell also takes as input the output of the previous execution (h_{t-1}).

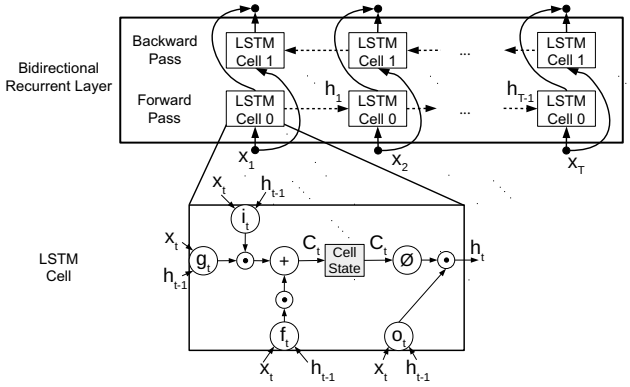

Fig. 2. Bidirectional LSTM layer and architecture of an LSTM cell.

$$i_t = \sigma(W_{ix}x_t + W_{ih}h_{t-1} + b_i) \quad (3)$$
$$f_t = \sigma(W_{fx}x_t + W_{fh}h_{t-1} + b_f) \quad (4)$$
$$g_t = \phi(W_{gx}x_t + W_{gh}h_{t-1} + b_g) \quad (5)$$
$$o_t = \sigma(W_{ox}x_t + W_{oh}h_{t-1} + b_o) \quad (6)$$
$$c_t = f_t \odot c_{t-1} + i_t \odot g_t \quad (7)$$
$$h_t = o_t \odot \phi(c_t) \quad (8)$$

Fig. 3. Computations performed in an LSTM cell. $\odot$, ϕ and σ are element-wise multiplication, hyperbolic tangent and sigmoid function respectively.

The bottom of Figure 2 shows the architecture of an LSTM cell, whereas Figure 3 provides the computations performed inside the cell. The key component is the cell state, c_t, that represents the memory storage for the cell. The cell state is updated by using four gates. The input gate (Equation 3) decides which information will be added to the cell state. The forget gate (Equation 4) decides which information will be removed from the cell state. The cell updater gate (Equation 5) provides candidate information to be added to the cell state. Finally, the output gate (Equation 6) decides which information of the cell state will be use to generate the output h_t. Each gate is implemented as an FC layer taking two different inputs: x_t, a.k.a. feed-forward input, and h_{t-1}, a.k.a. recurrent input. FC layers for the different gates take most of the computations in an LSTM network. Section III shows that both feed-forward and recurrent inputs exhibit a high degree of similarity across consecutive executions of the LSTM cell.

III. Temporal Reuse

This section analyzes the similarity between inputs of consecutive DNN executions in different layers, and introduces a technique that exploits this similarity to save computations and memory accesses. Table I shows the DNNs employed for the analysis. *Kaldi* is an MLP for acoustic scoring, a key task of a speech recognition system. It takes as input a window of 9 frames of speech (current frame and the four previous and four next frames), where each frame is represented as an array of 40 features. *Kaldi* generates as output the likelihoods of the 3482 senones, where a senone represents part of a phoneme.

TABLE I

DEEP NEURAL NETWORKS EMPLOYED FOR THE ANALYSIS OF COMPUTATION REUSE. THE TABLE ONLY INCLUDES FULLY-CONNECTED (FC), CONVOLUTIONAL (CONV) AND BIDIRECTIONAL LSTM (BILSTM) LAYERS, AS THESE LAYERS TAKE UP THE BULK OF COMPUTATIONS IN DNNS. OTHER LAYERS, SUCH AS RELU OR POOLING, ARE NOT SHOWN IN THE TABLE FOR THE SAKE OF SIMPLICITY.

Kaldi [5]: MLP for Acoustic Scoring (18MB)
Baseline Accuracy: 89.51%, Quantization Accuracy: 89.04%

Layer	Input Dim	Output Dim	Computation Reuse
FC1	360	360	-
FC2	360	2000	-
FC3	400	2000	75%
FC4	400	2000	66%
FC5	400	2000	56%
FC6	400	3482	66%

EESEN [10]: RNN for Speech Recognition (42MB)
Baseline Accuracy: 69.03%, Quantization Accuracy: 68.85%

Layer	In Dim	Out Dim	Cell Dim	Comp. Reuse
BiLSTM1	120	640	320	38%
BiLSTM2	640	640	320	53%
BiLSTM3	640	640	320	56%
BiLSTM4	640	640	320	59%
BiLSTM5	640	640	320	60%
FC1	640	50	-	-

C3D [8]: CNN for Video classification (300MB)
Baseline Accuracy: 94.86%, Quantization Accuracy: 93.48%

Layer	Input Dim	Output Dim	Kernel	Comp. Reuse
CONV1	3x16x112x112	64x16x112x112	3x3x3	-
CONV2	64x16x56x56	128x16x56x56	3x3x3	76%
CONV3	128x8x28x28	256x8x28x28	3x3x3	75%
CONV4	256x8x28x28	256x8x28x28	3x3x3	75%
CONV5	256x4x14x14	512x4x14x14	3x3x3	73%
CONV6	512x4x14x14	512x4x14x14	3x3x3	80%
CONV7	512x2x7x7	512x2x7x7	3x3x3	80%
CONV8	512x2x7x7	512x2x7x7	3x3x3	87%
FC1	8192	4096	-	88%
FC2	4096	4096	-	61%
FC3	4096	101	-	54%

AutoPilot [9]: CNN for Self-Driving Cars (6MB)
Baseline Accuracy: 99.69%, Quantization Accuracy: 99.63%

Layer	In Dim	Out Dim	Kernel	Comp. Reuse
CONV1	3x66x200	24x31x98	5x5	46%
CONV2	24x31x98	36x14x47	5x5	84%
CONV3	36x14x47	48x5x22	5x5	93%
CONV4	48x5x22	64x3x20	3x3	94%
CONV5	64x3x20	64x1x18	3x3	88%
FC1	1152	1164	-	89%
FC2	1164	100	-	97%
FC3	100	50	-	95%
FC4	50	10	-	82%
FC5	10	1	-	-

On the other hand, *EESEN* is an RNN for end-to-end speech recognition. It takes as input a sequence of frames, where each frame is represented as 120 features, and it generates the likelihoods of the 50 different characters in the target language for each frame in the input sequence. In short, *EESEN* is able to spell the words directly from audio frames.

Regarding the convolutional networks, *C3D* is a CNN for classifying actions in videos. The input consists of a non-overlapping window of 16 consecutive frames, i.e. the 16 frames for each CNN execution are disjoint. The CNN identifies the action that corresponds to that segment of the video, generating as output the likelihoods for all the possible 101 actions. Finally, *AutoPilot* is a CNN for self-driving cars that maps raw pixels from a single front-facing camera to steering commands.

We have evaluated the relative difference of the inputs in consecutive DNN executions for multiple layers. We found that, on average, the relative difference is lower than 14% for the DNNs shown in Table I. Figure 4 shows the relative difference in the inputs of the last two FC layers of *Kaldi* for a test audio file. Consecutive inputs tend to be similar, with a relative difference that ranges between 5% and 25%.

In this paper, we define **input similarity** as the percentage of inputs of a DNN layer that have not changed with respect to the previous execution of the DNN. Despite the small relative differences in the inputs, most of the 32-bit floating-point point values are not exactly the same than in the previous execution and, therefore, input similarity is small according to our strict definition. However, input quantization can be employed as an effective solution to increase the degree of similarity, exposing more redundant computations with a negligible impact on the accuracy.

Linear quantization [16] is a highly popular technique to map a continuous set of values to a discrete set. In this paper, we apply uniformly distributed linear quantization to the inputs

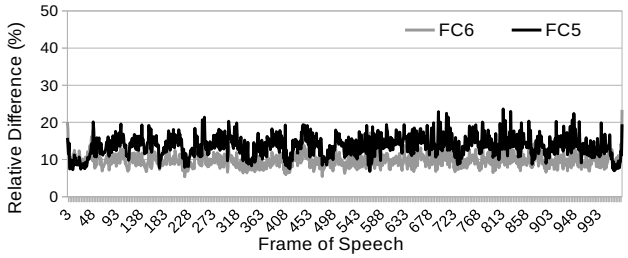

Fig. 4. Relative difference defined as the Euclidean distance between current and previous input vectors, divided by the magnitude of the input vector in the previous execution.

of the FC, convolutional and recurrent layers of DNNs. These layers represent close to 100% of the total execution time for typical neural networks. Then, we analyze the degree of similarity between the inputs of consecutive DNN executions, and the accuracy loss due to the quantization. For each input i and each layer l, the quantization is applied according to the following equation:

$$Qval_c = round\left(\frac{input_i}{step_l}\right) * step_l \quad (9)$$

The step is computed as the range divided by the number of clusters (C). The range of the inputs of a layer is obtained via profiling using the training dataset. $Qval_c$ is the cluster centroid that is closest to the original value of the input. The benefit of the quantization is that it significantly increases the similarity among the inputs of consecutive DNN executions, with a negligible impact in accuracy if enough clusters are provided.

In order to exploit the similarity in the inputs, the key observation is that, if the previous output of the neuron

is saved, inputs that have not changed with respect to the previous execution do not require any computation, since their contribution to the output of the neuron has not been modified. In case an input changes, the previous output can be corrected according to the following equation:

$$z_o^{'} = z_o + \sum_{i=1}^{N} \Big((c_i^{'} - c_i) * W_{io} \Big) \quad (10)$$

For each layer, we first compute the difference of the previous quantized input and the current one ($d_i = c_i^{'} - c_i$). For each output neuron o, the weight associated to input i is multiplied by its corresponding difference d_i and added to the previous output, only if d_i is different from zero. Otherwise, nothing needs to be done. If the percentage of inputs that remain unmodified is high, i.e. if the input similarity is high, then computing the output of a neuron in this manner requires significantly fewer computations and memory accesses.

In this paper, we define the **degree of computation reuse** as the percentage of computations that can be reused from the previous DNN execution, because their input operands have not been modified. Computation reuse and input similarity are highly related, since in case an input is the same as in the previous execution the previous results are reused avoiding all the computations associated with that input.

We analyzed the input similarity between frames of our four DNNs, shown in Table I, using multiple configurations: number of clusters, range of the inputs and layers where the quantization is applied. First, we analyzed the accuracy loss when applying linear quantization to different layers of the DNN. We started by quantizing the inputs of all the layers using 32 clusters, and observed that the accuracy is highly affected, mainly due to the first layers as their errors are propagated across the entire DNN. Due to this accuracy loss, we selectively applied the quantization layer by layer starting from the last layer, except for *EESEN* and *AutoPilot* where we start from BiLSTM5 and FC4 respectively, since the last FC layers of these networks are fairly small. After testing the accuracy of the DNNs when the quantization is applied to the last layer, we tested the accuracy when the technique is applied to the last two layers, and so on until finding the optimum number of layers where the quantization can be applied with negligible loss in accuracy.

After selecting the layers where the quantization is applied, we analyzed the impact of the number of clusters, using configurations with 8, 12, 16 and 32 clusters for the linear quantization. We found that linear quantization with 8 and 12 clusters introduces significant accuracy loss in all the DNNs. The configuration with 16 clusters achieves a small accuracy loss in *Kaldi* and *EESEN*, whereas 32 clusters are required in *C3D* and *AutoPilot* to maintain accuracy. When using quantization, the smaller the number of clusters the higher the similarity, since the inputs are constrained to a smaller set of values, but the larger the accuracy loss since input errors are increased. We found that the configuration with 16 clusters provides the best trade-off between similarity and accuracy

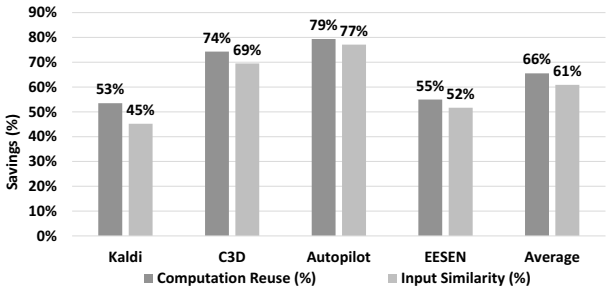

Fig. 5. Input similarity and compute reuse for our set of DNNs.

loss for *Kaldi* and *EESEN*, whereas the configuration with 32 clusters provides the best results for the CNNs.

Table I includes the accuracy for the baseline DNNs and the accuracy when using linear quantization of the inputs. Furthermore, it provides the degree of computation reuse for the layers where linear quantization is applied. In *Kaldi*, quantization is applied to the last four FC layers, obtaining a degree of computation reuse that ranges between 56% (FC5) and 75% (FC3). The accuracy loss is as small as 0.47%. For *C3D*, quantization is used in all the FC and convolutional layers except for CONV1, achieving a computation reuse between 54% (FC3) and 88% (FC1) with an accuracy loss of 1.38%.

On the other hand, quantization is applied in all the layers except the last FC for *EESEN* and *Autopilot*. Note that the number of neurons in the output layers of these DNNs is extremely low and, therefore, the potential for avoiding computations is fairly small compared to the entire DNN. In *AutoPilot*, the accuracy loss is 0.06% and the degree of computation reuse is bigger than 80% in most of the layers. Regarding *EESEN*, the potential for avoiding computations is smaller, but still more than 50% of the computations can be reused across consecutive DNN executions for most of the recurrent layers, with an accuracy loss of just 0.18%.

Figure 5 shows the degree of computation reuse and the input similarity for the overall DNNs. When using linear quantization, more than 50% of the computations can be reused across DNN executions in all the DNNs. On average, 61% of the inputs remain unmodified with respect to the previous DNN execution and 66% of the computations can be avoided. Therefore, we can conclude that our computation reuse technique exhibits a high potential for avoiding computations for different DNN architectures (MLPs, CNNs and RNNs) and different applications (acoustic scoring, speech recognition, video classification and self-driving cars).

IV. Reuse-Based DNN Accelerator

This section describes how the high degree of computation reuse, characterized in Section III, can be exploited to improve the performance and energy efficiency of DNN inference. First, we present the main hardware components of our DNN accelerator. Next, we describe how FC, convolutional and

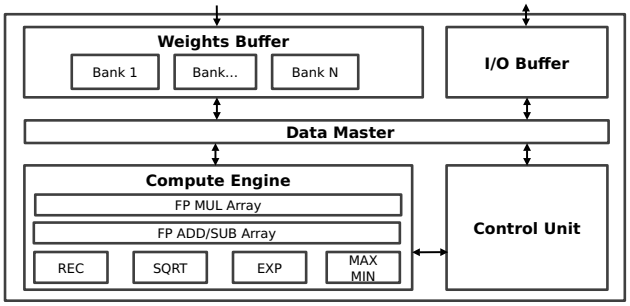

Fig. 6. Architecture of the DNN accelerator.

recurrent layers are executed in the accelerator using our computation reuse scheme.

A. DNN Accelerator Overview

In this paper, we present an accelerator for DNNs that takes advantage of the computation reuse found in MLPs, CNNs and RNNs. Our design exploits the high degree of similarity between consecutives inputs to save computations and memory accesses. A high-level block diagram of the accelerator is illustrated in Figure 6. The Compute Engine (CE) contains the functional units that perform the FP computations, including an array of FP multipliers and adders, together with specialized functional units (reciprocal, square root...). The CE is also employed to quantize the inputs in our reuse scheme and, hence, no extra hardware is required for quantization.

On the other hand, the Control Unit (CU) provides the appropriate control signals in every cycle. The CU contains the configuration of the DNN. In addition, it stores the centroids of the clusters employed for the quantization of the inputs of each layer.

Regarding the on-chip storage, the accelerator includes an eDRAM memory to store the synaptic weights of the different layers. Note that the memory footprint for the weights is typically quite large for common DNNs and, hence, significant on-chip storage is required to avoid excessive off-chip memory traffic that is very costly from an energy point of view [17]. eDRAM is used in our design to provide larger on-chip capacity with a small cost in area compared to SRAM. This memory is highly multi-banked to achieve the required bandwidth to feed a large number of functional units in the CE.

On the other hand, the I/O Buffer is an SRAM memory with two banks used to store the intermediate inputs/outputs between two DNN layers. Finally, the Data Master is in charge of fetching the corresponding weights and inputs from the on-chip memories, and dispatching them to the functional units in the CE.

The accelerator is configured for different DNNs by loading the neural network model that includes the information of each layer, i.e. input neurons, output neurons, weights and biases. In case all the parameters fit in the on-chip memories of the accelerator, the entire dataset is loaded from main memory. Otherwise, the accelerator loads the amount of elements that can fit on-chip, whereas the rest will be loaded from main memory on demand. By doing so, weights stored on-chip are reused across multiple DNN executions and, hence, no additional off-chip memory access is required for those elements. The accelerator is power gated during idle periods and, hence, weights are loaded from main memory at the beginning of processing every sequence of inputs (audio utterance, video...).

The vast majority of computations for MLPs, CNNs and RNNs come from FC, convolutional and recurrent layers respectively. Next subsections provide more insights on how these layers are executed in the accelerator with our computation reuse technique.

B. Computation Reuse in FC Layers

The FC layer computes the dot product of the inputs and the weights of each neuron (see section II-A). Figure 7 illustrates the execution of an FC layer in the accelerator when using the reuse scheme. Note that the first execution is different, as it computes the DNN from scratch whereas subsequent executions reuse previous results. The top of Figure 7 shows how the weights for an FC layer are interleaved in the Weights Buffer. That is, the first weight of every neuron is stored first, then the second weight of every neuron and so on. This layout simplifies the implementation of the computation reuse scheme, as it is simpler to locate all the weights that operate with a given input in case they have to be skipped or accessed to perform corrections.

The I/O Buffer is organized in three different areas. The first area stores the indices, i.e. the quantized inputs, as these values are required to verify whether an input remains unchanged with respect to the previous execution. The second area stores the outputs of all the layers where the computation reuse is exploited, to be later reused by the next DNN execution. Finally, the third area is used as a temporal storage for the inputs/outputs of layers where our reuse scheme is not applied. The first two areas are extra storage required to exploit computation reuse, Section VI shows that it represents a very small overhead.

The first execution of an FC layer is as follows: initially, the accelerator reads and quantizes the first input. In parallel, M weights of different neurons are read from the eDRAM. The index of the input quantization is stored in the I/O Buffer to be consumed in the next execution. Then, the accelerator performs M MULs of the input by the weights, followed by M ADDs to accumulate the result of each output neuron. Outputs are then stored in the I/O Buffer, to be used by the next layer and to be reused by the same layer in the next DNN execution. The accelerator is pipelined so, in the same cycle, the I/O Buffer reads one input, the Data Master reads M weights from memory and the CE performs M multiplications and M additions.

Subsequent executions of the FC layer employ our computation reuse scheme. The flowchart in Figure 7 illustrates the execution of an FC layer with computation reuse. Initially, the first input and its corresponding index from the previous execution are read from the I/O Buffer. Next, the current input is quantized, while the index is used to fetch the corresponding

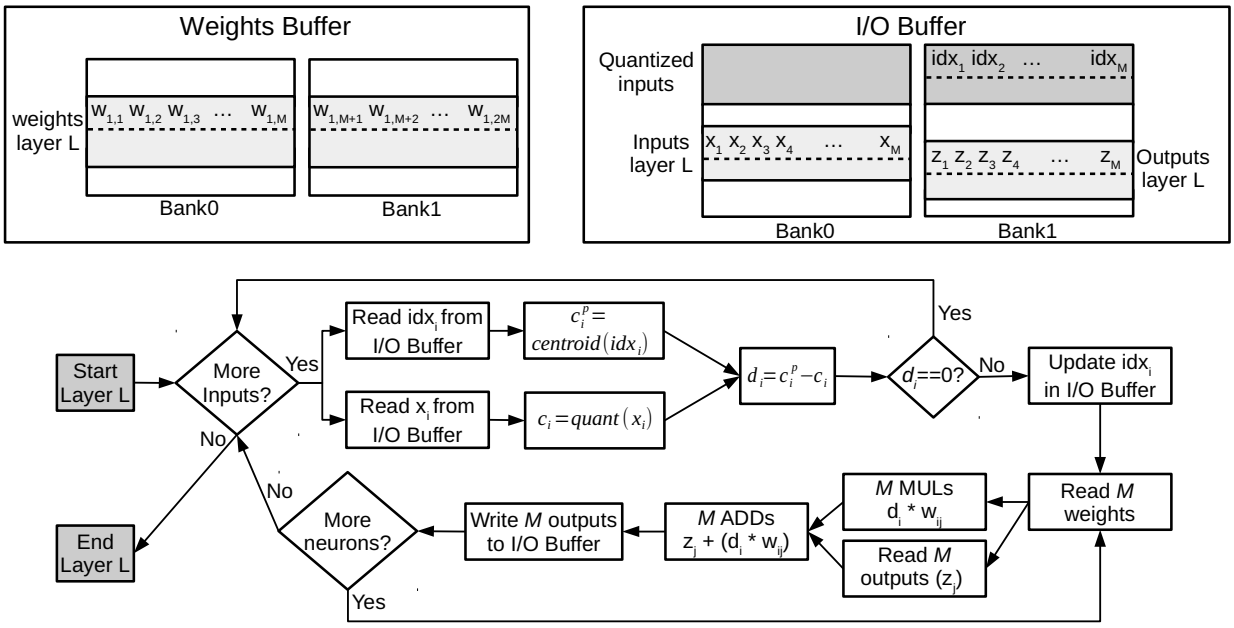

Fig. 7. FC Execution in the Reuse Accelerator.

centroid. The current quantized input is subtracted from the centroid. If the outcome is zero, the input is ignored and all the corresponding computations and memory accesses are skipped. In case the input has changed, the index is updated in the I/O Buffer and all the neurons are corrected using the weights associated to that input. Since the accelerator stores the outputs of the previous execution, corrections can be performed by removing the contribution of the previous input and adding the contribution of the current input. To this end, M weights from different neurons are fetched from memory. Then, M MULs compute the value that has to be added, multiplying the weights by the difference between the current input and the previous one. At the same time, M previous outputs are read from the I/O Buffer. Finally, the FP adders are employed in order to update the outputs, and the corrected results are updated in the I/O Buffer. This process is repeated until all the neurons are corrected for each modified input.

C. Computation Reuse in CONV Layers

In a convolutional layer, the weights of a kernel are shared among different inputs. Assuming a kernel of size $k \times k$ and f output filters, $k \times k \times f$ multiplications and additions are required for each input in the conventional scheme. Figure 8 illustrates the convolutional layer execution when using the reuse scheme. The dimensionality of the convolutional layers may be quite large, as shown in Section III. We employ a blocking scheme to reduce the on-chip storage requirements. The I/O Buffer only stores one block for each input feature map and one block for each output feature map, the block size being significantly smaller than the dimensions of the feature maps. In addition, our computation reuse scheme requires storage for the indices of the inputs in the previous DNN execution.

For the first execution, the accelerator loads one block of inputs and processes one input of the block at a time by multiplying it by the weights of the kernels that correspond to that input, and adding the results to the corresponding output neurons. The indices of the quantization and the output results of the convolutional layer are stored in the I/O Buffer and sent to main memory at the end, to be later used by the next CNN execution.

The flowchart in Figure 8 illustrates the subsequent CNN executions exploiting computation reuse. Initially, for each input channel, an input block and its corresponding quantized indices are read from main memory and written to the I/O Buffer. Then, the corresponding output block of each output feature map is read from main memory. Once the current blocks are loaded on-chip, the first input and its corresponding index are read from the I/O Buffer. Then, the input is quantized, while the index is used to access the centroid. The current quantized input is subtracted from the centroid. If the outcome is zero, the input is skipped, avoiding all the corresponding computations. In case the input has changed, the index is updated in the I/O Buffer and the output neurons that operate with that input are corrected. Assuming a kernel size of $k \times k$ with stride one, the accelerator will update $k \times k$ output neurons for each filter. To this end, the accelerator reads $k \times k$ weights, multiplies them by the difference between the current and previous input, while loading $k \times k$ previous outputs from the I/O Buffer. Finally, the FP adders are employed to update the outputs for the current execution, and the updated results are written in the I/O Buffer. The indices and output blocks are written back to main memory

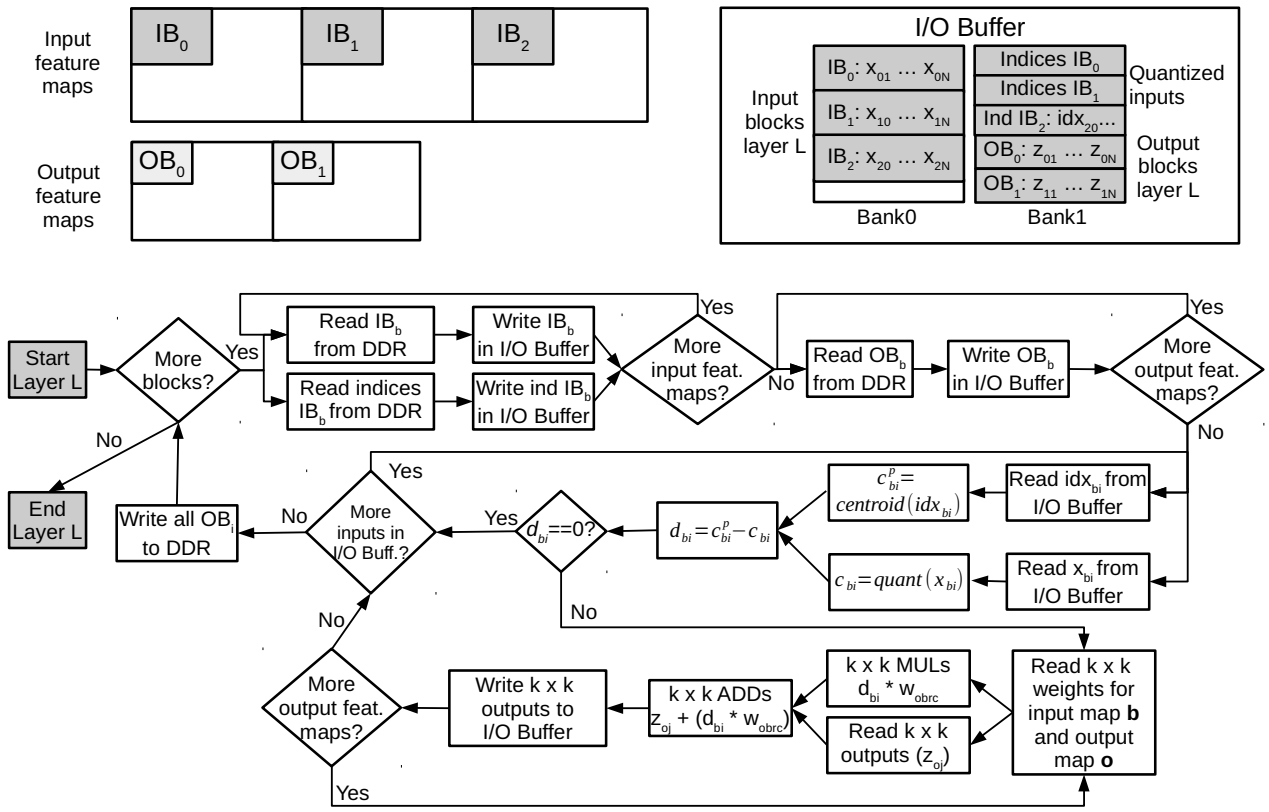

Fig. 8. Execution of a convolutional layer in the accelerator. IB_b means Input Block from input feature map b, whereas OB_o means Output Block from output feature map o.

at the end of each input block. This process is repeated until all the neurons have been updated for each modified input.

D. Computation Reuse in Recurrent Layers

A recurrent layer contains one (unidirectional) or two (bidirectional) LSTM cells that are recurrently executed for each element in the input sequence. An LSTM cell, described in Section II-C, consists of four gates implemented as four independent FC layers. Therefore, the execution of a recurrent layer mimics the behavior of FC layers in the accelerator, i.e. each gate is processed as described in Section IV-B. However, there are a few differences that make recurrent layers more amenable for our computation reuse technique. First, each recurrent layer is executed back-to-back for every input in the sequence before proceeding to the next layer (see Figure 2). Therefore, RNNs only require extra storage for the inputs/outputs of one layer, whereas MLPs and CNNs require extra storage for all the layers where the computation reuse technique is applied. In other words, temporal locality of the redundant computations is higher in RNNs. Second, the four gates (four FC layers) in one LSTM cell share the same inputs. Hence, we only compare the inputs once with the previous values and, in case an input remains unmodified, computations and memory accesses are avoided in the four gates.

E. Accelerator with Multiple Tiles

Multiple instances of the accelerator shown in Figure 6 can be integrated in the same chip to improve performance and accommodate large DNNs. Each instance of the accelerator, or tile, includes a router to communicate results with the other tiles. The different tiles are connected in a ring. The workload is distributed as follows. For FC layers, output neurons are evenly distributed among the tiles. Regarding convolutional layers, the different filters are distributed among the tiles. Finally, for recurrent layers, different tiles process different gates of one LSTM cell.

V. Evaluation Methodology

We have developed a simulator that accurately models the hardware accelerator presented in Section IV, including the computation reuse scheme. Table II shows the parameters for the experiments. We model an accelerator with four tiles, with a total of 128 FP adders and 128 FP multipliers (32 per tile). Our accelerator includes 36 MB of eDRAM for the Weights Buffer (9 MB per tile), which is enough to fit on-chip two of the evaluated DNNs (i.e. *Kaldi* and *Autopilot*), whereas for *C3D* it can store most of the convolutional weights and for *EESEN* it stores the weights of one layer at a time. Finally, the I/O Buffer is sized to fit the blocked input of the *C3D* DNN,

TABLE II
PARAMETERS FOR THE ACCELERATOR.

Technology	32 nm
Frequency	500 MHz
# of Tiles	4
# of 32-bit multipliers	128
# of 32-bit adders	128
Weights Buffer	36 MB
I/O Buffer Size	1152KB (Baseline) / 1280KB (Reuse)

which requires 1152 KB for the baseline and 1280 KB when using the reuse scheme, as extra storage is needed for the input indices (see Figure 8). We employ a block size of $16 \times 16 \times 1$ for the CNNs, as we found that it provides a good trade-off between on-chip storage requirements and memory bandwidth usage. Regarding main memory, we model an LPDDR4 of 4 GB with a bandwidth of 16 GB/s (dual channel).

Regarding area and energy consumption, the combinational logic hardware is implemented in Verilog and synthesized to obtain the delay and power using the Synopsys Design Compiler, the modules of the DesignWare library and the technology library of 28/32nm from Synopsys [18]. For the technology library, we use the standard low power configuration with 0.78V. On the other hand, we characterize the memory components of the accelerator by obtaining the delay, energy per access and area using CACTI-P [19]. We use the configurations optimized for low power and a supply voltage of 0.78V. Finally, the energy consumption of main memory is estimated by using the MICRON power model for LPDDR4 [20]. The results obtained with the aforementioned tools are combined with the activity factors provided by our simulator to obtain the dynamic and static power of each hardware component.

Our objective is to prove that our scheme provides important savings for multiple applications and different DNN architectures. To this end, we evaluate our technique on four state-of-the-art DNNs from different application domains, including acoustic scoring, speech recognition, video classification and self-driving cars. We use the DNNs shown in Table I. We include an MLP for acoustic scoring implemented in the Kaldi [21] toolkit, a popular framework for speech recognition, trained with Librispeech [22] dataset. In addition, we employ *C3D*, a CNN from Facebook [8] that is implemented in Caffe [23]. To evaluate *C3D*, we use videos from the UCF101 benchmark suite [24]. We also use *Autopilot*, which is a CNN for self-driving cars from NVIDIA [9] implemented in Tensorflow [25], [26]. We employ as input videos from a single front-facing camera mounted behind the windshield of a car. Finally, we include an RNN for end-to-end speech recognition implemented in EESEN [10] and trained with TED-LIUM dataset [27]. For all the DNNs, we employ several hours of audio/video to assess the accuracy and performance of our computation reuse scheme. Our workloads represent important machine learning applications, and the selected DNNs cover the three existing approaches for sequence processing: MLPs, CNNs and RNNs.

The different deep learning frameworks used (Kaldi, Caffe, Tensorflow and EESEN) provide state-of-the-art software implementations for DNNs on CPUs and GPUs. The GPU versions are implemented in CUDA using highly-optimized libraries such as cuBLAS or cuDNN. We compare our accelerator with the software implementations running on a high performance CPU, an Intel i7 7700K (Skylake), and a modern high-end GPU, an NVIDIA GeForce GTX 1080 (Pascal). We use the RAPL library [28] to collect energy consumption of the CPU, and nvidia-smi (`NVIDIA System Management Interface`) [29] to measure GPU power dissipation.

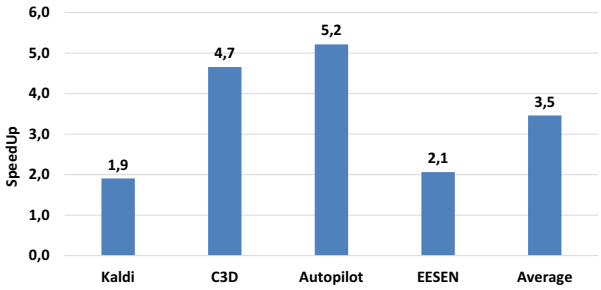

Fig. 9. Speedups achieved for each DNN. Baseline configuration is the DNN accelerator without the computation reuse technique.

VI. EXPERIMENTAL RESULTS

This section evaluates the performance and energy consumption of our computation reuse scheme. First, we present the speedups and energy savings achieved by our technique when implemented on top of the accelerator presented in Section IV. Next, we characterize the memory overheads of the computation reuse scheme. Finally, we compare the accelerator with software implementations running on a modern CPU and GPU.

Figure 9 shows the speedups achieved by the computation reuse scheme. Our technique provides consistent speedups for the four DNNs that range from 1.9x (*Kaldi*) to 5.2x (*Autopilot*), achieving an average performance improvement of 3.5x. The reduction in execution time is due to reusing previously computed results, since inputs that remain unmodified with respect to the previous DNN execution do not require any computation or memory access. Furthermore, the overhead of performing the quantization and comparing the current input with the previous one is fairly small, since it is performed per input and not per connection. As one input is normally used in thousands of neurons, performing a comparison to detect that the input has not changed can save thousands of computations and memory accesses. *C3D* and *Autopilot* exhibit the highest degree of computation reuse (see Figure 5) and, hence, they obtain the largest performance improvements.

Figure 10 reports normalized energy. On average, our scheme reduces energy consumption of the accelerator by 63%. The energy savings are well correlated with the degree of input similarity and computation reuse reported in Figure 5. These energy savings are due to two main reasons. First, dynamic energy is reduced due to the savings in computations and

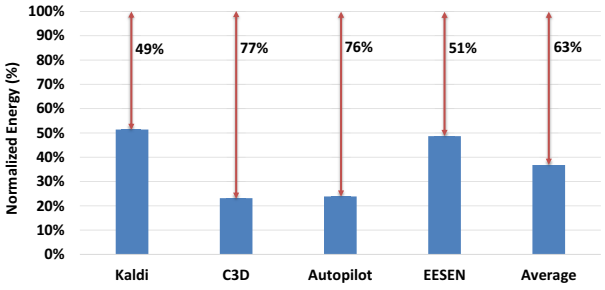

Fig. 10. Normalized energy for each DNN. Baseline configuration is the DNN accelerator without the computation reuse technique.

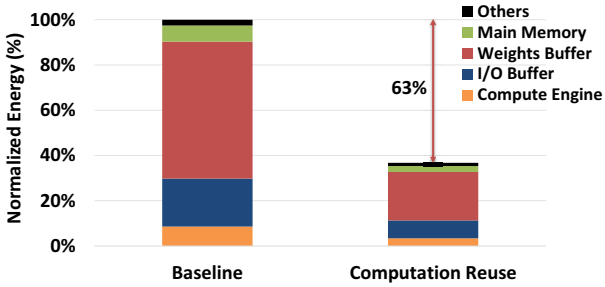

Fig. 11. Energy breakdown for the baseline accelerator and the version using the computation reuse scheme.

memory accesses. Second, the performance improvements shown in Figure 9 provide a reduction in static energy. Again, *C3D* and *Autopilot* obtain the largest benefits, achieving a reduction of 77% and 76% in energy respectively.

Figure 11 shows the energy breakdown of the DNN accelerator, including the percentage of energy consumed by each hardware component for the baseline accelerator (Baseline) and the version using our reuse scheme (Computation Reuse). The figure shows aggregated results for the four neural networks. As it can be seen, the eDRAM for storing the weights consumes most of the energy in both versions. The energy savings achieved by our reuse scheme are significant in all components, and are especially large in the on-chip eDRAM memory, since our scheme provides significant savings in memory accesses for fetching the synaptic weights.

On the other hand, the reduction in the number of computations also results in smaller energy in the Compute Engine. Note that the energy required for performing the quantization and comparing the current and previous inputs for every DNN execution is also included in the Compute Engine energy consumption. Regarding the I/O Buffer, its energy consumption is significantly reduced since inputs that remain unmodified do not require any access to this on-chip memory.

Table III reports the memory overheads of the computation reuse technique. For *Kaldi* (MLP), the reuse scheme requires extra storage in the I/O Buffer for the indices (quantized inputs) and outputs of the different layers (see Figure 7). For *EESEN* (RNN), our accelerator requires storage for the inputs/outputs of the four gates of one LSTM cell in the I/O Buffer. Note that in the worst case only 66 KB of on-chip storage are required for *Kaldi* and *EESEN*, whereas no additional storage is used in main memory. Regarding the CNNs (*C3D* and *Autopilot*), the I/O Buffer requires additional storage for the indices (see Figure 8). The inputs/outputs of each layer are stored in main memory for the CNNs, resulting in a small increase (around 10%) in storage requirements as reported in Table III. We provision the I/O Buffer with enough capacity for the largest DNN (*C3D*): 1152 KB in the baseline and 1280 KB with the reuse scheme. In addition, 1.25 KB are used for the table of centroids. The overall overhead in area of the accelerator is less than 1%, as it increases from 52 mm^2 to 53 mm^2.

TABLE III
MEMORY OVERHEADS OF REUSE SCHEME.

DNN	I/O Buffer (KB)		Main Memory (MB)	
	Baseline	Reuse	Baseline	Reuse
Kaldi	27	66	18	18
C3D	1152	1280	397	443
Autopilot	160	176	6.6	7.2
EESEN	8	13	42	42

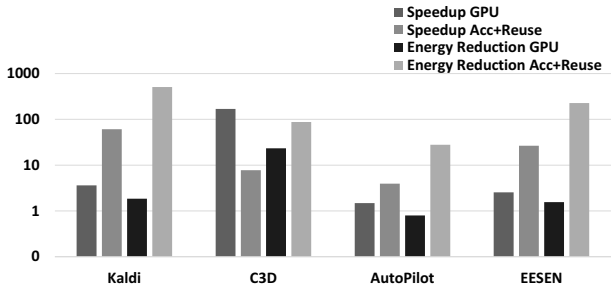

Fig. 12. Speedup and energy reduction. Baseline configuration is the Intel i7 7700K CPU.

Our simulator and power model include all the overheads due to the computation reuse technique: extra accesses to a bigger I/O Buffer, extra memory bandwidth usage and memory storage, accesses for fetching centroids and computations for applying linear quantization. As it is shown in Figure 9 and Figure 10, these overheads are negligible in comparison to the savings in computations and memory accesses, and the net result is an improvement of 9.5x in energy-delay (2.7x in energy and 3.5x in delay).

Finally, Figure 12 provides a comparison of our accelerator including computation resue (Acc+Reuse) with a modern CPU (i7 7700K) and a recent GPU (GTX 1080). Regarding the speedups, our accelerator outperforms both CPU and GPU in all the DNNs except in *C3D*. *C3D* is the largest DNN and it achieves close to peak performance in the GPU. Note that the GTX 1080 exhibits the highest peak performance, as it includes 2560 FPUs at 1.82 GHz versus the 256 FPUs at 500 MHz of the accelerator. However, the GPU also exhibits the highest power dissipation, bigger than 200 W for *C3D*.

Regarding energy consumption, the accelerator provides large energy reductions with respect to both CPU and GPU for all the DNNs. On average, the energy reduction over the i7 7700K and the GTX 1080 is 213x and 115x respectively.

A. Reduced-precision Accelerator

Reduced-precision fixed-point arithmetic has become popular in DNN accelerators [11]. In this work, we evaluate our computation reuse technique on top of an accelerator that employs 32-bit floating-point arithmetic (see Section IV-A). Nevertheless, our technique also provides large improvements in performance and energy consumption when using reduced-precision arithmetic. Note that our scheme is already using quantized inputs and, hence, the input similarity and computation reuse should not be significantly affected when changing from 32-bit floating-point to 8-bit fixed-point. The use of reduced-precision should benefit both the baseline system and the version using our computation reuse scheme in the same or similar amount. Therefore, we expect relative numbers, such as speedup or normalized energy, to be very similar to the ones reported in the paper when using a reduced-precision accelerator as the baseline. To support this claim, we modified our DNN accelerator to employ 8-bit fixed-point arithmetic, using 8 bits to represent both weights and inputs, and we evaluated its performance for the Kaldi DNN. We found that the input similarity increases from 45% (32-bit floating-point baseline) to 52% (8-bit fixed-point baseline), whereas we obtain a large amount of computation reuse of 58%. Our scheme provides 1.8x speedup and 45% energy savings for Kaldi DNN when implemented on top of a reduced-precision accelerator. Furthermore, we also verified that the accuracy loss is negligible (significantly lower than 1%).

VII. Related Work

This section outlines state-of-the-art accelerators for DNNs and techniques to optimize these accelerators.

Machine Learning. DNNs have become highly popular in a wide range of environments and devices, from large data centers and high performance computers [1], [30] to mobile devices [31], [32]. DNNs are computationally intensive and consume a significant amount of energy. Therefore, custom architectures can improve both aspects. Some of the most popular applications of MLPs are speech recognition and machine translation [12], [33], whereas CNNs are commonly used for object recognition, image classification [3] and, more recently, video classification and labeling [34], [35]. Finally, RNNs [10], [36] represent a promising solution for sequence processing problems, as they can store information of past inputs to improve accuracy of future predictions.

Custom Accelerators. A reference work for DNN accelerators is the DianNao [37], and their variations DaDianNao [1] for large scale and ShiDianNao [32] for mobile devices. Although there are several proposals before DianNao [38]–[40], those are highly restricted to small neural networks such as MNIST, and rely too much in accessing main memory, which represents an important overhead in energy consumption when applied to larger DNNs. DianNao was the first accelerator proposal that includes its own on-chip SRAM buffers to reduce the memory accesses, and DaDianNao further improved this aspect by adding eDRAM to store the weights. Most of the accelerators presented in recent years are based in DianNao, such as Minerva [4]. The most recent works on accelerators are focused on low power [2], [4], [32]. Common techniques to reduce power include the reduction of the voltages of some components in exchange of reliability [4], reducing the width of the functional units and data to fixed point precision [41] or pruning [2] the neural network to reduce the amount of computations and consequently the energy consumption. Our work is different from these previous accelerators as we propose to exploit computation reuse across multiple DNN executions to save energy and improve performance. To the best of our knowledge, this is the first proposal that exploits computation reuse based on input similarity in DNNs.

Computation Reuse. There have been many studies that observed the inherent fault tolerance of DNNs [4], [38]. This characteristic can be used to trade accuracy for higher performance or energy savings. Other studies also observed that many of the computations done in a neural network share at least one value and, thus, some computations can be reused with a low impact in accuracy [42], [43]. However, these works are focused on exploiting computation reuse in a short time span, i.e. during the same execution of a layer of the DNN, and the reuse is applied considering only the weights of the convolutional and FC layers. In contrast, our accelerator reuses the computations from one execution of the DNN to the next, and applies the reuse taking into account the similarity found in the inputs of each layer, which is the main contribution of this paper.

VIII. Conclusions

In this paper, we show that modern DNNs usually process a sequence of data (e.g., frames of audio or video) and consecutive neuron's inputs exhibit a high degree of similarity for all neurons, including hidden layers. We propose a new mechanism that exploits this similarity, in order to avoid a large percentage of the activity by reusing the results of the previous execution. We analyze the degree of input similarity for several DNNs and observe that, if linear quantization is used, on average more than 60% of the inputs of fully-connected, convolutional and recurrent layers remain unmodified with respect to the previous execution. Based on this observation, we propose a hardware extension for DNN accelerators to exploit this similarity.

The proposed reuse scheme checks the inputs that have changed with respect to the previous execution. Inputs that remain unmodified are ignored, avoiding the associated computations and memory accesses, whereas modified inputs are used to correct the previous output result of each neuron in that layer. We implement our reuse scheme on top of a state-of-the-art DNN accelerator. We show that our technique requires minor changes, mainly additional memory storage for saving the outputs of the DNN layers. Our experimental results show

that, on average, our scheme provides 63% energy savings and 3.5x speedup, while it only requires a minor increase in the area of the accelerator (less than 1%). Our scheme works for MLPs, CNNs and RNNs from different applications, including speech recognition, video classification and self-driving cars.

ACKNOWLEDGMENT

This work was supported by the Spanish State Research Agency under grant TIN2016-75344-R (AEI/FEDER, EU). The main author work is supported by the Spanish Ministry of Education under grant FPU15/02294.

REFERENCES

[1] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun *et al.*, "Dadiannao: A machine-learning supercomputer," in *IEEE 47th MICRO*, 2014.

[2] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: efficient inference engine on compressed deep neural network," in *IEEE 43rd ISCA*, 2016.

[3] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer cnn accelerators," in *IEEE 49th MICRO*, 2016.

[4] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernández-Lobato, G.-Y. Wei, and D. Brooks, "Minerva: Enabling low-power, highly-accurate deep neural network accelerators," in *IEEE 43rd ISCA*, 2016.

[5] X. Zhang, J. Trmal, D. Povey, and S. Khudanpur, "Improving deep neural network acoustic models using generalized maxout networks," in *IEEE ICASSP*, 2014.

[6] R. Yazdani, J.-M. Arnau, and A. González, "Unfold: A memory-efficient speech recognizer using on-the-fly wfst composition," in *IEEE 50th MICRO*, 2017.

[7] H. Tabani, J.-M. Arnau, J. Tubella, and A. Gonzalez, "An ultra low-power hardware accelerator for acoustic scoring in speech recognition," in *IEEE 26th PACT*, 2017.

[8] D. Tran, L. Bourdev, R. Fergus, L. Torresani, and M. Paluri, "Learning spatiotemporal features with 3d convolutional networks," in *IEEE ICCV*, 2015.

[9] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang *et al.*, "End to end learning for self-driving cars," *arXiv*, 2016.

[10] Y. Miao, M. Gowayyed, and F. Metze, "Eesen: End-to-end speech recognition using deep rnn models and wfst-based decoding," in *IEEE ASRU Workshop*. IEEE, 2015.

[11] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," in *IEEE/ACM 44rd ISCA*, 2017.

[12] J. Zhang and C. Zong, "Deep neural networks in machine translation: An overview," *IEEE Intelligent Systems*, 2015.

[13] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, 1998.

[14] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey *et al.*, "Google's neural machine translation system: Bridging the gap between human and machine translation," *arXiv*, 2016.

[15] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, 1997.

[16] B. Widrow, I. Kollar, and M.-C. Liu, "Statistical theory of quantization," *IEEE Transactions on instrumentation and measurement*, 1996.

[17] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both weights and connections for efficient neural networks," *CoRR*, 2015.

[18] "Synopsys," https://www.synopsys.com/, accessed: 2017-07-20.

[19] S. Li, K. Chen, J. H. Ahn, J. B. Brockman, and N. P. Jouppi, "Cacti-p: Architecture-level modeling for sram-based structures with advanced leakage reduction techniques," in *IEEE/ACM ICCAD*, 2011.

[20] "Micron lpddr4 system power calculator," https://www.micron.com/support/tools-and-utilities/power-calc, accessed: 2017-07-20.

[21] D. Povey, "Kaldi software," http://kaldi-asr.org/, accessed: 2017-07-20.

[22] V. Panayotov, G. Chen, D. Povey, and S. Khudanpur, "Librispeech: an asr corpus based on public domain audio books," in *IEEE ICASSP*, 2015.

[23] "Caffe software," http://caffe.berkeleyvision.org/, accessed: 2017-07-20.

[24] "Ucf101 action recognition benchmark," http://crcv.ucf.edu/data/UCF101.php, accessed: 2017-07-20.

[25] "Tensorflow framework," https://www.tensorflow.org/, accessed: 2017-07-20.

[26] "Autopilot-tensorflow," https://github.com/SullyChen/Autopilot-TensorFlow, accessed: 2017-07-20.

[27] A. Rousseau, P. Deléglise, and Y. Esteve, "Ted-lium: an automatic speech recognition dedicated corpus." in *LREC*, 2012.

[28] V. M. Weaver, M. Johnson, K. Kasichayanula, J. Ralph, P. Luszczek, D. Terpstra, and S. Moore, "Measuring energy and power with papi," in *41st International Conference on Parallel Processing Workshops*, 2012.

[29] NVIDIA, "NVIDIA System Management Interface," https://developer.nvidia.com/nvidia-system-management-interface, accessed: 2017-07-20.

[30] W. Xiong, J. Droppo, X. Huang, F. Seide, M. Seltzer, A. Stolcke, D. Yu, and G. Zweig, "Achieving human parity in conversational speech recognition," *CoRR*, 2016.

[31] J. Jin, V. Gokhale, A. Dundar, B. Krishnamurthy, B. Martini, and E. Culurciello, "An efficient implementation of deep convolutional neural networks on a mobile coprocessor," in *IEEE 57th MWSCAS*, 2014.

[32] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "Shidiannao: Shifting vision processing closer to the sensor," in *ACM/IEEE 42nd ISCA*, 2015.

[33] G. E. Dahl, T. N. Sainath, and G. E. Hinton, "Improving deep neural networks for lvcsr using rectified linear units and dropout," in *IEEE ICASSP*, 2013.

[34] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *22Nd ACM International Conference on Multimedia*, 2014.

[35] A. Karpathy, G. Toderici, S. Shetty, T. Leung, R. Sukthankar, and L. Fei-Fei, "Large-scale video classification with convolutional neural networks," in *IEEE CVPR*, 2014.

[36] F. Silfa, G. Dot, J. Arnau, and A. González, "E-PUR: an energy-efficient processing unit for recurrent neural networks," *CoRR*, 2017.

[37] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. 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*, ser. ACM ASPLOS, 2014.

[38] O. Temam, "A defect-tolerant accelerator for emerging high-performance applications," in *39th ISCA*, 2012.

[39] A. Hashmi, H. Berry, O. Temam, and M. Lipasti, "Automatic abstraction and fault tolerance in cortical microachitectures," in *38th ISCA*, 2011.

[40] A. Hashmi, A. Nere, J. J. Thomas, and M. Lipasti, "A case for neuromorphic isas," in *ACM ASPLOS*, 2011.

[41] P. Judd, J. Albericio, and A. Moshovos, "Stripes: Bit-serial deep neural network computing," *IEEE Computer Architecture Letters*, 2017.

[42] J. Wu, C. Leng, Y. Wang, Q. Hu, and J. Cheng, "Quantized convolutional neural networks for mobile devices," *CoRR*, 2015.

[43] A. Yasoubi, R. Hojabr, and M. Modarressi, "Power-efficient accelerator design for neural networks using computation reuse," *IEEE Computer Architecture Letters*, 2017.