Re-architecting the On-chip memory Sub-system of Machine-Learning Accelerator for Embedded Devices

Ying Wang, Huawei Li, and Xiaowei Li
State Key Laboratory of Computer Architecture
Institute of Computing Technology, Chinese Academy of Sciences, Beijing, P.R. China
{wangying2009, lihuawei, lxw}@ict.ac.cn

ABSTRACT

The rapid development of deep learning are enabling a plenty of novel applications such as image and speech recognition for embedded systems, robotics or smart wearable devices. However, typical deep learning models like deep convolutional neural networks (CNNs) consume so much on-chip storage and high-throughput compute resources that they cannot be easily handled by mobile or embedded devices with thrifty silicon and power budget. In order to enable large CNN models in mobile or more cutting-edge devices for IoT or cyberphysics applications, we proposed an efficient on-chip memory architecture for CNN inference acceleration, and showed its application to our in-house general-purpose deep learning accelerator. The redesigned on-chip memory subsystem, Memsqueezer, includes an active weight buffer set and data buffer set that embrace specialized compression methods to reduce the footprint of CNN weight and data set respectively. The Memsqueezer buffer can compress the data and weight set according to their distinct features, and it also includes a built-in redundancy detection mechanism that actively scans through the work-set of CNNs to boost their inference performance by eliminating the data redundancy. In our experiment, it is shown that the CNN accelerators with Memsqueezer buffers achieves more than 2x performance improvement and reduces 80% energy consumption on average over the conventional buffer design with the same area budget.

1. INTRODUCTION

Deep convolutional neural networks (CNN) are making substantial progress in computer vision, image processing, speech recognition and other Recognition, Mining and Synthesis (RMS) applications. The advent of deep learning architecture is expected to enable machine intelligence in lightweight devices such as mobile phones, robotics and micro unmanned vehicles. Researchers on deep learning and architecture are putting emphasis on GPGPU-based optimization or designing specialized accelerators to deal with deep neural networks, which is well-known for its demand for large computational and storage capability. Due to their numerous network parameters and large working-set, embracing typical CNN models for machine intelligence is not easy in low-end devices with limited resource and power budget. How to decrease the memory footprint of CNN invocation without compromising the strength of the learning model is a key step to enable real-time machine learning in embedded, IoT or cyberphysical devices. Conventionally, there are a lot of software-based compression methods proposed to promote memory utility of CNNs in CPUs or GPGPUs [1].

ICCAD '16, November 07-10, 2016, Austin, TX, USA
 ISBN 978-1-4503-4466-1/16/11…$15.00
DOI: http://dx.doi.org/10.1145/2966986.2967068

These software approaches either use general data compression based on sparse matrix and quantization [1], or leverage ad-hoc techniques to eliminate the structural or functional redundancy in CNN models, which are essentially machine learning techniques [1]. Software approaches are proved to work well for CPU and GPU, however, they are not as efficient as hardware compression in specialized CNN accelerators for multiple factors [2]:

First, conventional methods are oblivious to the underlying architecture of CNN accelerators, and they are unable to address the bandwidth under-utilization caused by data misalignment or random memory access. Sparse matrix based schemes are bound to be variable-length compression to achieve higher compression ratio, which cause data misalignment in the buffers of accelerator. It causes unpredictable access latency in data moving that destroys the continuous data streaming between memory and processing elements (PEs) in such typical streaming accelerators that will suffer from poor random memory access performance [1] [2] [3], and eventually leads to an underutilization of bandwidth and performance degradation. Second, conventional software compression are unaware of balance between on-chip memory and the computation throughput of PEs because it often takes multiple cycles to recover the operands from their compressed form. This breaches one important principle in hardware design: Roofline model. According to Roofline model, the data bandwidth has to be in perfect balance with the throughput of processing elements. However, it is non-trivial to achieve balance between the wide PE set and the compressed data stream in CNN accelerators. Due to the varied compression ratio of each data unit (weight or feature data), the actual bandwidth utility is changing while the PE throughput keeps constant, making it hard to achieve the balance. Last of all, software compression is insufficient to eliminate the value redundancy in the work-set of CNNs, and it can be accompanied by dynamic hardware compression for even higher compression ratio.

For CNN accelerators, a specialized on-chip memory architecture with hardware compression could be more efficient to squeeze large CNNs into lightweight devices, and keeps them free of the issues faced by software based compression. In this work, we propose an active on-chip memory design, Memsqueezer, for low-overhead deep learning accelerators. In general, Memsqueezer is different from prior general purpose compression methods like sparse matrix or quantization:

First, active buffer is intended to work for CNN accelerators based on ASIC or FPGA, and it does not bring in data misalignment in the memory arrays. It ensures that the accelerator only has simple data streaming patterns and a simple address generation mechanism for better energy-efficiency.

Second, Memsqueezer buffer design takes the work-set categorization of CNN into account by resorting to different hardware compression policies. Because CNN includes multiple data sets: weight and data, Memsqueezer includes two parts: an active weight buffer that leverages off-line data clustering and base-delta partitioning method to compress weight set, and active data buffers that dynamically compress and decompress the intermediate data stream with a frequent-pattern based method. Prior CNN compression methods are prone to compress the

neural weight, but seldom seek solutions on intermediate data compression. We found that intermediate data can also be the bottleneck of on-chip storage when the weight data is compressed at a high ratio. Worse still, ignoring intermediate data compression can also impair the balance between memory bandwidth and computation throughput once the uncompressed data stream occupies more bandwidth than weight and becomes the new weak point.

Last of all, Memsqueezer integrates an in-memory redundancy detector to eliminate the all-zero or near-zero patterns in weight and intermediate data and fast forward computation operation with zero-pattern input to PEs, so that data compression can not only benefit memory utility but also lead to performance benefits.

Conclusively, Memsqueezer greatly shrinks the on-chip memory size and boosts the performance of CNN inference. With Memsqueezer, the CNN accelerators will become more lightweight, and more bandwidth efficient, enabling large-scale CNNs in resource-constrained devices.

A Motivational study

A typical CNN consists of multiple inter-connected convolutional layers, pooling layers and full connection layers, which are usually processed serially in an inference accelerator. The convolutional and full connection layers often involve a large set of weights, which put a high pressure on on-chip storage and bandwidth. The data decomposition of several popular CNN models are shown in Table 1. It shows some of the sampled layers for state-of-the-art CNN models. It is seen that the work-set is so large that prior accelerators often integrates a big and expensive on-chip buffer to avoid off-chip memory access [2] [3]. Assuming Alexnet is used for image recognition, if the net is mapped to a low-power FPGA-based accelerator, e.g. low-end Zynq-7020 SoC, we must at least achieve 5x~ data compression ratio in the on-chip block memory. Otherwise, frequent DRAM accesses will emerge and greatly harm the QoS of on-line image recognition measured in FPS. The other point indicated in Table. 1 is that intermediate data can also become the bottleneck of on-chip storage. For example, even Squeezenet requires an up to 2310K on-chip data buffer to avoid buffer misses. Therefore, Memsqueezer is to provide an active on-chip memory subsystem with a higher capacity and bandwidth utility so that both weight and data can be handled in low-overhead CNN accelerators without inducing frequent buffer misses.

Table. 1 Weight and data set size for different layers of CNN models

	Layer	Conv1	Conv2	Conv3	Conv5	Fc1	FC2
Alexnet	data (K)	294	567.2	364.5	126.7	18	8
	weight(K)	68	600	1728	864	73728	32768
VGG	data(K)	**6272**	1568	2048	256	64	8
	weight(K)	72	144	1152	4608	262144	32768
Squeezenet	data(K)	2310	576		364.5	-	-
	weight(K)	-	162	-	-	-	-

2. Related Work

Deep learning Accelerator Deep convolutional neural networks relates to intensive convolution operations between a bunch of feature maps (data) and pre-trained kernel filters (weight), thus matrix convolution and multiplication takes most of the computation time in CNN inference [2] [3] [4]. Early CNN accelerators are focused on data-path optimization. [2] exploits data-level parallelism within feature maps and convolution kernels, but it cannot scale to various network and layer types. There are also a lot of memory-centric accelerators or processors proposed to exploit data locality inside the convolutional kernels of varied size and stride [5] [6]. These CNNs accelerators put a lot of emphasis on memory optimization for locality preservation, and they often adopt a large on-chip memory and rely on the continuous data streaming between memory and PEs to achieve high throughput, which cannot be ruined by compression methods.

CNN Compression In prior study, compressing the working-set of CNN are mostly done at the application level. Deep compression [1] reduces the storage overhead to run large networks by means of pruning, quantization and Huffman coding to compress the weights. Unlike the Memsqueezer, deep compression preliminarily focuses on weight pruning and allow the identical weight values to be shared by multiple connections. These techniques are supposed to work for convolutional neural network invoked on GPGPUs or general purpose CPUs. As to hardware-based compression, there are also sparse-matrix based techniques proposed to skip the unnecessary computation for CNN acceleration purpose [7] [8] [9], which are conducted at the same time with this work. EIE is thought as a hardware implementation of deep compression, and it effectively accelerate sparse matrix-vector multiplication in CNN inference [8]. Eyeriss exploits zero valued neurons for memory compression, but not for computation speed-up. In contrast to sparse-matrix based methods, Memsqueezer is a specialized hardware-level compression for mobile or embedded deep learning accelerators (DLAs) or processors. It is a simple re-design of on-chip memory sub-system combined with two different compression methods, and also fully conscious of the balance between the streaming bandwidth and PE throughput in DLAs.

3. Architecture of Memsqueezer

3.1 The overall architecture of active buffers for DLA

As depicted in Fig. 1, a typical data-centric CNN accelerator [2] [3] [6] consists of several major components: data buffers for input and output data, a weight buffer, processing elements (PEs) that carry out multiply-and-accumulate and other arithmetic operations. In addition, there is a coordinator (omitted in Fig. 1) that generates the control signals to the PEs, and AGUs that generates the continuous address to fetch both weight and data into PEs. Except hardware, a compiler is often required to translate the network specification, including the information like layer count, layer type, kernels size, kernel count and active-function type, into control streams (instructions) for the coordinator. Once the instructions are ready, the input data and weights of pre-trained model are continuously streamed into PEs driven by the AGUs.

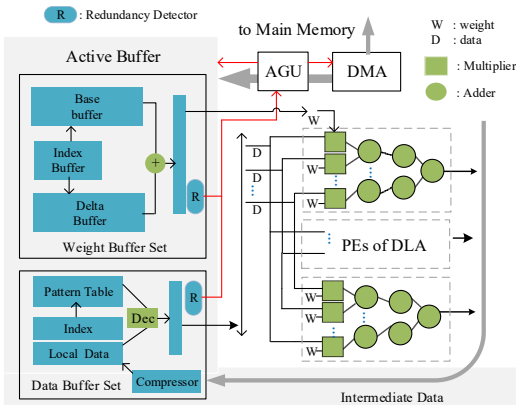

Fig. 1 Diagram of Memsqueezer buffers in a typical CNN accelerator

Due to the large work-set of weights and data as illustrated in Table. 1, the size of data and weight buffers have to be large enough to avoid long-latency data fetch requests to the off-chip memory. Such large buffers as in prior work [3] [5] is thought unaffordable in mobile or embedded systems. In our active memory shown in Fig. 1, a weight buffer is partitioned into one or multiple base buffers, an index buffer, and one delta buffer to store the weights in a compressed and well-aligned form. In addition to the weight buffer set, the data buffers are used to store the feature data and the intermediate data, which is produced by one CNN layer and then consumed by the next layer in a streaming way. Different from the weight buffer set, the data buffer set includes a pattern table and a local buffer working in coordination. The AGUs generates the continuous address offsets to the index buffers, and

drive they to finally produce the uncompressed weight and data for the high throughput PEs. Besides, a small bias buffer used to store network bias is omitted in Fig. 1. In later chapters, the bias is assumed to be treated the same way as weight if specified otherwise.

3.2 Memsqueezer Weight Buffer Design

a. The structure of active weight buffer

Weights are generated at the off-line stage of net training, and they are read-only data that could be preprocessed before network invocation. Thus, even a compression algorithm with high complexity and overhead will not impact the performance by amortizing the running overhead at off-line stage. Generally, Memsqueezer weight buffer adopts a *base+delta* method to partition the weights into a small set of base and delta values that could be shared by all weights throughout the layers. When the preprocessed base set and delta set are loaded into the base buffers and delta buffers, they are used to generate the real weights as illustrated in Fig. 2. As a result, the large volume of weights can be reduced into a small batch of bases and deltas, greatly decreasing the footprint of weight buffer.

Fig. 2 illustrates the basic structure of weight buffer set, including an index table, a base buffer, and a delta buffer. In each cycle, all these three buffers are indexed once, and output the uncompressed weight partitions that match the issuing width of the PEs. Here a weight partition includes *n* weight words equal to the width of *n* PEs.

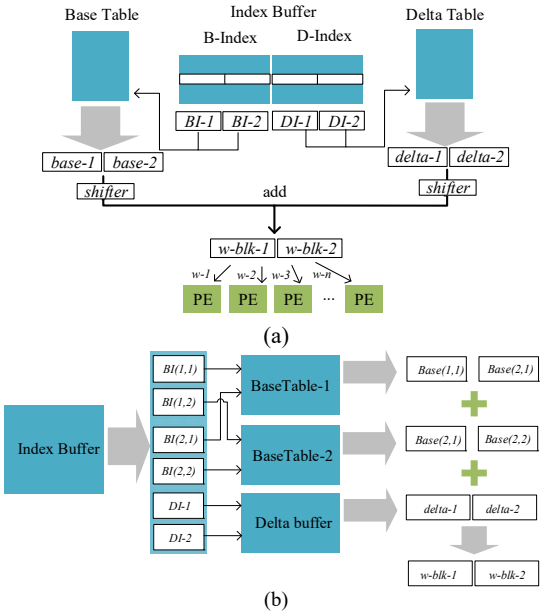

Fig. 2 Organization of Memsqueezer weight buffer set

Though the size of weight partition is set to match the width of PE, Memsqueezer does not necessarily compress the *n*-sized weight partition as a whole, or compress each of *n* weight words separately (16-bit or 32-bit). The compression granularity is between *1~n* words and still to be decided to achieve the highest compression ratio in later sections. Here we define that a block is the basic unit to be compressed. Fig. 2 (a) only shows an example when the *n*-sized weight partition is divided into two blocks: *w-blk-1* and *w-blk-2*. These two blocks are obtained by adding the *base-1/base-2* to *delta-1/delta-2* respectively. Bases are selected from the base table by Base Index (BI) while deltas are selected from the delta table by Delta Index (DI). Both of BI and DI lies at the same row of the index buffer and they are streamed out of index buffer under the command of the weight AGU.

Except compression granularity, the number of bases and deltas also matters in terms of the achievable compression rate. Memsqueezer does not necessarily store the weight partition in the form of *base+delta*. Memsqueezer can re-divide the weight so that it takes the form of (*base-1+base-2+…+base-n+delta)*. In Fig. 2 (b), a wider index buffer contains multiple groups of BI and DI. In Fig. 2 (b), BI(*i,j*) represents the *j*-th bases of the compressed block-*i.* In this way, the uncompressed weight block *w-blk-i* becomes the sum of multiple bases and the delta.

The basic organization and working mechanism of active weight buffer set has been introduced, but the technical details about how to select the bases and deltas will be discussed in next section.

b. Preprocessing the weights into deltas and bases

How to partition the weights into bases and deltas have great impacts on the compression ratio. In order to use the minimum number of bases and deltas to represent the target weight set, we uses a K-means based method to preprocess the weight data set.

As shown in Fig. 3, K-means clusters the scattered weight values into *K* compact domains. Each point indicates a weight block to be compressed. Each domain has a centroid with minimum summed distances to all other points in this domain. Such a centroid is used as the base value, and all other weight points in that domain are likely to have a small difference (*delta*) from the centroid (*base*) value. All of the weight words that are to share the common base can then be represented by a compact *base+delta* form.

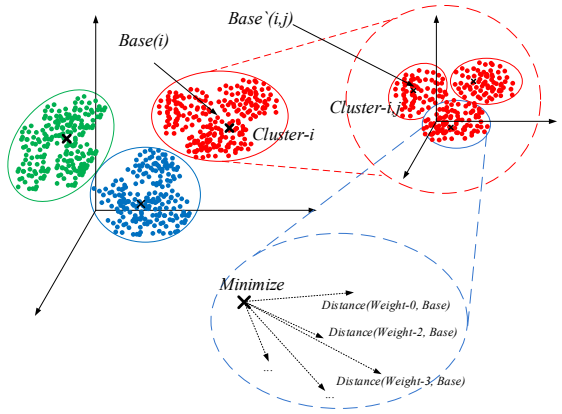

Fig. 3 Procedure of base and delta generation

One important parameters in K-means is *K*. Clustering the weights into *K* domains means that there will be *K* base entries in the base table of active weight buffer, which directly decided the compression rate. In this work, we conduct an exhaustive study of *K* selection to see how many bases are needed to represent the weights at least. As we discussed in last section, there could be more than one base needed to reproduce one weight block. Therefore it takes repetitive rounds of K-means clustering to generate the multiple bases. As shown in Fig.3, for cluster-*i* obtained by the first round of K-means, the value of centroid (*base(i)*) will be subtracted from each points in this cluster, creating a new cluster of *delta* points defined as cluster-*i'*. After subtraction, the new points in this *delta* cluster-*i'* will be re-clustered by another round of K-means to generate multiple new clusters. For any new cluster-*i,j* that is a subset of large cluster- *i'*, its centroid, base(*i, j*), will be used as the second base for points in cluster- *i,j*. In this way, all the points (weights) in cluster-*i,j* can be presented in the form of *{base(i)+ base(i,j)+delta}*. K-means clustering can also be repeated for many more times to generate more bases and so leads to more base tables. We also explore the design space to find the base table size and weight partition size that achieves the highest compression ratio in experiments.

Supposing that *N* weight blocks are compressed at the granularity of *a-bit, k* bases and *m* deltas at the width of *b*-bit are used, we will have an index table as large as $N \cdot (log\ k + log\ m)$, the final compression ratio of the Memsqueezer weight buffer will be:

$$\frac{N\cdot(log\ k+log\ m)+b(k+m)}{N\cdot a} \quad (1)$$

3.3 Active Data buffer design

Similar base-delta compression cannot be applied to the active data buffers for two reasons: 1. Data are generated dynamically, and it degrades performance significantly to use an on-line approach to compress the dynamic data into the form of deltas and bases. 2. The featured distribution of values in data and weight are not the same.

Therefore, we refer to a fast and inexpensive on-line frequent-pattern based method to compress the feature data set [10]. Frequent pattern based compression assumes that a block of output data consist multiple repetitive short patterns like 0x0000 and it is a waste to store the full length words output by the *n* PEs. Instead, if we store the most frequently encountered short patterns like 0x00001111 (halfword) in a compact table, and the output data can be represented by the sequence of pattern indexes, the space will be greatly shrunk.

Due to the sparsity of neural activation data, frequent pattern based compression can reduce the length of output words generated by *n* PEs. Some of the neuron output words are all zeros, or contains multiple continuous zero bits, but they are normally stored in a full word. By using a frequent pattern index to represent the small values, half-zero or full-zero word will shorten the word length. For example, many small-value values are generated by the activation or pooling layers can be stored in 6-bit, 4-bit or even shorter words. After compression, each word is divided into a prefix used to indicate pattern types pre-stored in a dictionary and a local word that consists of multiple condensed segments within the value range specified by the related patterns. The simplest pattern table is described in Table. 2 that exemplifies only four patterns. The table shows the indexes stored in the index table of active buffer set, and the size of local data in local buffer after compression. In experiments, we use four more patterns by partitioning the value range into finer intervals for better compression results.

Table 2. Patterns in the dictionary table

Patterns	index	Size of local data
all zero	00	2 bit
Half word, -16< *v* <16	01	5 bit
Upper half word, the lower ending 4 bits in a word are all zero	10	5 bit
all other patterns	11	16 bit

Fig. 4 describes the compression and decompression stages in the data buffers. The data buffer set includes a pattern index table and a local data buffer. Whenever the PEs output *n* words, they are compared to the patterns in the parallel pattern comparator that determines which pattern the words belong to respectively. Simultaneously, the result is sent to the encoder that shortens the *n* words separately into localized data falling into the range specified by its matched pattern. Then their pattern indexes are put continuously into the pattern index table, while the shortened words are put into the local data buffer accordingly. This is the complete procedure of on-line data compression.

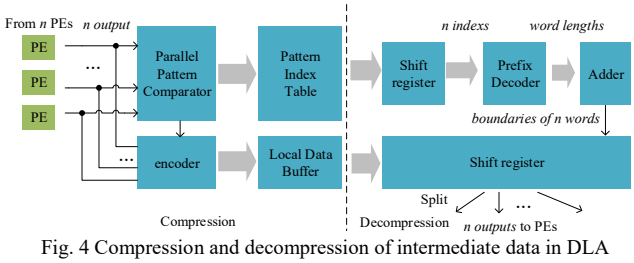

Fig. 4 Compression and decompression of intermediate data in DLA

In the right part of Fig. 4, it is shown that under the control of data AGU, decompressor firstly uses a shift register to split the output index row from index table into multiple segment pattern indexes, which will be then input to a pattern decoder. The pattern decoder translates the pattern indexes into the value ranges for the separate *n* local words. The outputs of pattern decoder are added up to find the starting offsets and ending offsets for all the local data words stored in the local data buffer. With the splitter offsets, the output row from the local data buffer can be correctly split in the shift register into *n* uncompressed full length words for the PEs [10].

3.4 Eliminating the computation redundancy with Memsqueezer

Memsqueezer also brings the opportunities of performance boost by pre-annotating the redundant values (all zero patterns or near-zero patterns) in both weight and data set. If the active buffers can give correct feedbacks to the AGUs and let it skip the unnecessary operations with zero input, there will be considerable gains. According to our analysis, there exists a 40-70 % proportion of zero operands in the feature maps and weights of all layers in CNNs.

For data buffer, skipping the zero data that will not make any changes to the accumulate result of convolutional layers is relatively simple. The zero or half zero patterns are reserved in the pattern table that can be particularly used to indicate the zero operand in computation. With frequent pattern based compression in the data buffer, manifesting that the data segment contains zero words is much earlier than decoding other patterns. For example, when the data is compressed at the granularity of two words, pattern index 0011 in the pattern table will indicates the compressed block contains a non-zero word and a zero word skippable in PEs.

However for active weight buffers, it is more complex to detect non-zero words since the weight is compressed in a base+delta way. The zero words have to be detected after the final uncompressed weights are generated. When the weight block is generated, the built-in detector will compare it to zero to generate the zero-mask for PEs. As shown in Fig. 5 (a), a zero-word detector is located next to the output latch of weight buffer, and it compares the row of weights with zero threshold (t_w) to generate the *n*-bit zero mask for the *n* PEs.

When all the *n*-bit zero masks for weight and data are respectively generated by the zero detector and the data buffer, another redundancy detection circuit is needed as described in Fig. 5 (b) to find enough zero word to form an issue window of *n* words, so that the corresponding issue window can be skipped in the *n* PEs to proceed to the next batch of weight and data partition. It is seen from Fig. 5 (b) that the *n*-bit zero masks of data and weight buffers are processed with a bit-wise *AND* operation, and the results from the first level AND gates are then inverted and all input to a AND gate that generates the single-bit redundancy indicator for computation skipping. Only if there are *n* continuous redundancy mask bits detected in either weight or data buffer as shown in Fig. 5 (b), the data and weights in next issue window can be completely evacuated. The final AND result is sent to inform both weight and data AGUs to issue the next *n* data-weight pairs to PEs.

Even when the current issue windows cannot be skipped due to the existence of non-zero input words, the mask signals can also be sent to AGUs to prevent loading the zero-data from either data or weight buffer, so that only the unmasked data-weight pairs are received by the according PEs while the other PEs are disabled, which is to save energy by disabling the unnecessary data load and computation.

As shown in Fig. 5 (a), comparators are used in the “zero-weight” detector to generate the zero-mask for weight. With the comparators, we also find a more aggressive way to boost energy-efficiency by marking the “sufficiently small values” as zero patterns, which trades off computation accuracy for energy-efficiency.

The threshold t_w is tunable for small number detection. Comparing the absolute value of weight to t_w means the input data smaller than t_w will also be dumped to skip the corresponding operation in the PEs According to the philosophy of approximate computing, small values

can sometimes be treated equally as noises to be filtered by activation functions, inducing minor or even no accuracy loss. To fully exploit the small numbers in various layers to skip more computation redundancy, we empirically configure threshold t_w to different values for the convolutional and full-connection layers respectively, ensuring no visible precision loss will be witnessed.

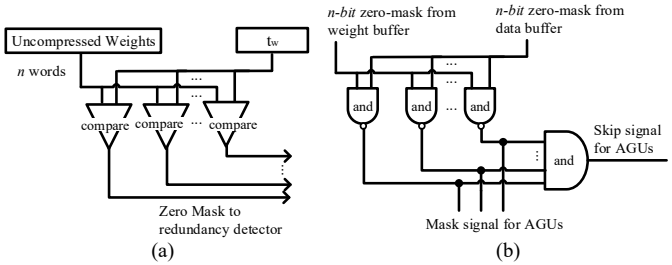

Fig. 5 Redundancy detectors in Memsqueezer buffers

3.5 The overhead of Memsqueezer buffers

Performance overhead There is no performance degradation induced by Memsqueezer because all compression and decompression operations are off the critical path of CNN inference. In Memsqueezer, the weight and data are continuously streamed out of buffers to PEs in deterministic time steps under the control of AGUs. The index address generation, the addition and shift operation related to data decompression can be pipelined and performed several cycles ahead of the according PE computation. Therefore, according to the Roofline model, only if the streaming bandwidth of weight and data buffers are equal to PEs' throughput, data decompression and PE computation can be carried out in a pipelined way with a bridging issue buffer in between , and both of memory and PEs will function at a 100% utility. In addition, Memsqueezer buffers have a much smaller footprint than normal buffers due to high compression ratio, and they can operate in a higher frequency than normal large buffers in [2] [3], so they are more likely to work in balance with the high-speed computation logics of PEs.

Hardware overhead The capacity of buffer is significantly decreased, but additional overhead has been raised due to the peripheral circuits of multiple buffers and table banks, frequent pattern based compression logic and the redundancy detection circuits. We implemented the compression logic of Memsqueezer buffers, redundancy detection circuit with 45nm process technology [11], and also evaluate the storage overhead of buffers with CACTI [15]. In experiment, we will fairly compare the Memsqueezer buffers and normal buffers in different set-ups [2] [3]

4. EVALUATION

4.1 Experimental Setup

We implemented the synthesizable RTL model of CNN accelerators integrated with Memsqueezer. The architecture of CNN accelerator is the consistent with the design in [12]. A compiler is responsible for processing the input of hardware configuration and CNN descriptive files (*.prototxt in Caffe) to generate on-line control bits and address patterns. In addition, we also built a cycle-accurate detailed simulator for fast simulation. The simulator includes a full function model that produces the inference output for various CNNs, and also a cycle accurate performance model that outputs the power/performance profile, which is built based on the synthesis result of the accelerator implemented with open-source 45nm process technology [11]. The hardware description of accelerator can be found in Table. 3. The operating frequency of the accelerator is 200Mhz and the voltage supply is 1.0*v*. Table. 3 also shows the total storage overhead of normal buffers and the Memsqueezer (MS) buffer set including all index table or other buffers. It can also be shown that Memsqueezer buffer offers a constant nominal bandwidth matched to that of PEs though the buffer port width is much narrower, which is a proof of high bandwidth utility. Block size in Table. 3 shows the granularity of MS compression whilst *Num.* shows the number of buffers used in weight and data buffer set.

For function level simulation, state-of-the-art neural networks are implemented with Caffe and trained in a server that contains four-way NVidia K40 GPU cards. The trained weight of CNNs are fed to the simulator to simulate network inference process.

Benchmark We evaluated several popular CNNs: Alexnet, GoogleNet , VGG and NiN as the workloads. The parameters of these networks are shown in Table 4. The input test dataset is from the ImageNet validation set [13]. The accelerators will execute the inference procedure of different CNNs that are to classify these random ImageNet pictures.

Table 3. Parameters of the implemented Accelerators

Name	Num.	Blocksize	TotalSize	Bandwidth
PE	16/32	-	-	2∗16∗(16or32)bit
Data-buf	2	-	2MB	16∗(16or32)bit
Data-buf (MS)	2 index 2 local buf	32-bit	290KB	<16∗(16or32)bit
Weight-buf	1	32-bit	1MB	16∗(16or32)bit
Weight-buf(MS)	2 base tables 1 delta buffer	32-bit	154KB	<16∗(16or32)bit
Bias_buf	1	-	1K	-

Table 4. Parameters of invoked CNNs

Network	Alexnet	Googlenet	VGG	NiN
Conv1 detail	3,11,4,96	3,7,2,64	3,3,1,64	3,11,4,96
#conv layers	5	57	16	12
Kernel types	11,5,3	7,5,3,1	3	11,5,3,1

4.2 Experimental Result

a) Performance

We compared the Memsqueezer with several counterparts. **MS** shows the performance of CNN accelerator with Memsqueezer buffer, while **Normal** indicates the inference performance of CNN accelerator with normal data and weight buffer described in Table. 3. **Equal-Norm** indicates the performance of normal data and weight buffers, which have the same area overhead as that of Memsqueezer buffers. CACTI is used to ensure that Equal-Norm has an equivalent overhead as that of Memsqueezer for fair comparison [14]. Last of all, **MS-RE** indicates the performance of Memsqueezer that activates the capability of redundancy detection in AGUs.

MS-Retrained is an optimized version of MS-RE that leverage the error-resilience feature of network retraining to achieve better compression and redundancy elimination results. For example, to have more compact delta tables, **MS-Retrained** intentionally rounds those close delta values up into one delta entry to shrink delta buffer. However, approximation induces additional accuracy loss in network inference. Thus, in MS-Retrained, an off-line network retraining step is invoked to compensate the accuracy losses and update the weights by network self-adaption. Meanwhile, small number checker used to skip the operations with near-zero input will also introduce precision loss in network inference. Similarly, MS-Retrained can also compensate the accuracy loss of small number approximation in off-line retraining stage. In this stage of approximation-aware network training, MS-Retrained tunes the value of weights to use less delta values and at the same time removes the weight of small values at feed-forward propagation stage. Afterwards, the back-propagation training will self-adapt the network parameters to compensate the potential accuracy loss. This is due to the intrinsic error resilience of neural network training.

Fig. 6 compares the inference time (x10^3 cycles) of the first convolutional layer, which projects the performance of the major computation task in CNN inference. Although with the same area budget, **MS** buffers have a much higher capacity utilization by compressing the work-set. Therefore, **MS** is much faster than **Equal-**

Norm. In general, the average performance of MS is close to that of Normal, but **1.6x** higher than Equal-Norm that suffers from frequent off-chip memory access due to the limited buffer size. MS-RE outperform MS by 11% on average while MS-Retrained outperform normal MS by 19% with the optimization of network retraining.

Fig. 7 shows the total all-layer network inference time (x10^6 cycles) of the compared schemes. Similarly, among all the schemes, **MS-Retrained** achieves the highest speed, which is **2.1** times faster than Equal-Norm on average. Meanwhile, MS and Normal still have very close performance. Compared to MS and Normal, MS-RE achieves 12% performance speed-up on average due to the skipped redundant computation with zero operands.

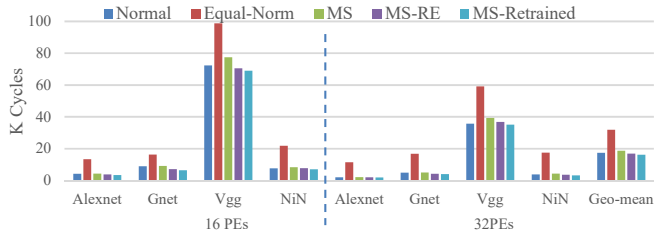

Fig. 6. Execution time of Conv-1 layer

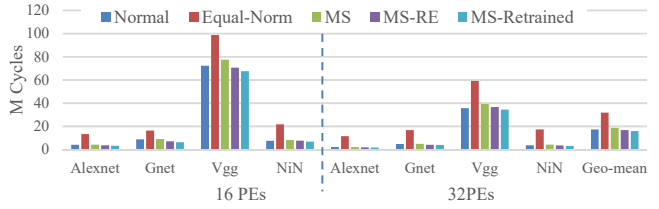

Fig. 7 Comparison of total inference time

b) *Energy consumption*

In Fig. 8, we compare the total energy consumption of the evaluated schemes. It is shown that **MS** consumes only 25% of the energy dissipated in Normal, while Equal-Norm consumes 190% more energy than Normal. The saved energy comes from several sources: 1. MS has more compact on-chip storage and narrow memory ports; 2. MS has better performance so that it takes less time to complete the network inference. Meanwhile, MS-RE achieves 22% energy reduction over MS since it does not only skip all the zero operand loading operation but also accelerate the computation by redundancy elimination. MS-Retrained consumes 26% less energy over that of MS. In the energy statistics, the power of index/pattern tables, the local/delta/base buffers, all the compression logic and redundancy detection logic are faithfully accounted for in the simulator.

c) *Precision Loss*

In order to achieve higher compression rate, Memsqueezer approximates the delta entries in the weight buffer and approximates the near-zero patterns as zero word in the aggressive-mode redundancy detector as introduced in last sections. Although we cautiously set the threshold t_w and the threshold of delta entry merging, these two factor might still cause precision loss in the evaluated CNNs. However, MS and MS-Re is configured to use only conservative zero-elimination strategy to avoid precision loss, whilst MS-Retrained uses an aggressive approach and higher threshold to merge delta values and define "zero" operand thanks to the protection of approximation-aware retraining. Therefore, we use the simulator to classify 50K random images from ImageNet with both MS-RE and Normal, and show their accuracy in Fig. 9. It is seen that the accuracy of MS-RE and MS are almost the same with that of Normal. For MS-Retrained, the classification accuracy only slightly lags (<1.6% on average) behind that of Normal due to the compensation of network retraining. In our experiments, it is seen that MS-Retrained can increase the compression ratio of weight buffer by 22% on average and improve the effects of redundancy-elimination by 15% with little precision losses.

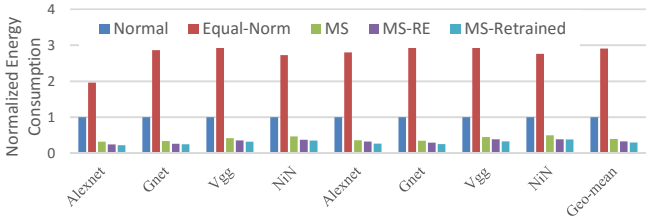

Fig. 8. Energy Consumption of network inference

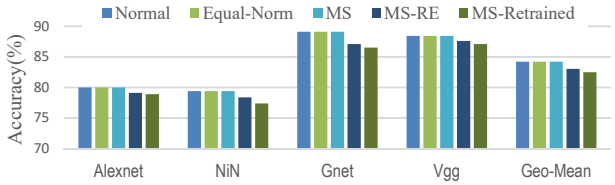

Fig. 9 Classification Accuracy comparison

5. CONCLUSION

Overall, we propose an efficient on-chip memory subsystem for low-overhead CNN accelerators, Memsqueezer, to squeeze the large CNN models into mobile or embedded devices. It is proved that the proposed Memsqueezer weight and data buffers can achieve higher capacity and on-chip bandwidth utility. The experiments shows that when compared to the normal buffers of the same area budget Memsqueezer can significantly boost the performance of typical CNN accelerators by 80%~130%, decrease the energy consumption by 83% on average. It is a very useful on-chip memory redesign that makes high performance CNN accelerators feasible in low-end devices.

6. ACKNOWLEDGMENTS

This work was supported in part by National Natural Science Foundation of China under Grant No. 61432017, 61176040, 61504153, 61402146 and 61521092. Huawei Li is the corresponding author.

REFERENCES

[1] S. Han, et al., "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," arXiv preprint, 2015.
[2] S. Cadambi, et al., "A programmable parallel accelerator for learning and classification," in Proc. *PACT*, 2010.
[3] T. Chen, et al., "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in Proc. of *ASPLOS*, 2014.
[4] Y. Wang, et al., "DeepBurning: Automatic Generation of FPGA-based Learning Accelerators for the Neural Network Family," in Proc. *DAC*, 2016.
[5] C. Farabet, et al., "NeuFlow: A runtime reconfigurable dataflow processor for vision," In CVPR Workshop, 2011.
[6] S. Chakradhar, et al., "A dynamically configurable coprocessor for convolutional neural networks," in Proc *ISCA*, 2010.
[7] J. Albericio, et al., "Cnvlutin: Ineffectual-Neuron-Free Deep Neural Network Computing," in Proc. *ISCA*, 2016.
[8] S. Han, et al., "EIE: Efficient Inference Engine on Compressed Deep Neural Network," in Proc. *ISCA*, 2016.
[9] Y. Chen, et al., "Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks," in Proc. *ISCA*, 2016.
[10] A. R. Alameldeen, et al., "Adaptive cache compression for high-performance processors," in Proc. *ISCA*, 2004.
[11] Nangate Open Cell Library. http://www.si2.org/openeda.si2.org /projects/nangatelib.
[12] L. Song, et al., "C-Brain:A deep learning accelerator that tames the diversity of CNNs through adaptive data-level parallelization," in Proc. *DAC*, 2016.
[13] O. Russakovsky, et al., "Imagenet large scale visual recognition challenge. International Journal of Computer Vision," pp.1-42, 2014.
[14] Cacti 6.5, in: http://www.hpl.hp.com/research/cacti/.