ZCOMP: Reducing DNN Cross-Layer Memory Footprint Using Vector Extensions

Berkin Akin*
Intel Labs, OR
berkin.akin@intel.com

Zeshan A. Chishti
Intel Labs, OR
zeshan.a.chishti@intel.com

Alaa R. Alameldeen
Intel Labs, OR
alaa.r.alameldeen@intel.com

ABSTRACT

Deep Neural Networks (DNNs) are becoming the prevalent approach in computer vision, machine learning, natural language processing, and speech recognition applications. Although DNNs are perceived as compute-intensive tasks, they also apply intense pressure on the capacity and bandwidth of the memory hierarchy, primarily due to the large intermediate data communicated across network layers. Prior work on hardware DNN accelerators leverages the cross-layer data sparsity via fully-customized datapaths. However, dynamically compressing/expanding such data is a challenging task for general-purpose multi-processors with virtual memory and hardware-managed coherent cache hierarchies.

In this paper, we observe that the DNN intermediate data is either sequentially streamed or reshaped with a regular transformation between layers. Hence, accesses to this data can tolerate a sequential or block sequential compression/expansion without requiring random element retrieval. Based on this insight, we propose ZCOMP, a CPU vector ISA extension tailored for DNN cross-layer communication. ZCOMP compactly represents zero value compression/expansion and fully automates the metadata generation, storage and retrieval which eliminates the need for several extra instruction executions and register usage. ZCOMP can be targeted both for inference and training to dynamically compress/expand cross-layer data before being written to memory. Our evaluations for individual layers and end-to-end DNN networks demonstrate that ZCOMP offers substantial data traffic reduction, both on-chip across cache-hierarchy and off-chip to DRAM, and performance improvements over no compression and existing AVX512 compression approaches.

CCS CONCEPTS

• **Computer systems organization → Parallel architectures**; *Single instruction, multiple data*; *Neural networks.*

KEYWORDS

Deep learning, memory system, sparsity, compression, ISA, CPU

ACM Reference Format:
Berkin Akin, Zeshan A. Chishti, and Alaa R. Alameldeen. 2019. ZCOMP: Reducing DNN Cross-Layer Memory Footprint Using Vector Extensions. In *The 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52), October 12–16, 2019, Columbus, OH, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3352460.3358305

*Now employed at Google.

MICRO-52, October 12–16, 2019, Columbus, OH, USA

ACM ISBN 978-1-4503-6938-1/19/10.
https://doi.org/10.1145/3352460.3358305

1 INTRODUCTION

In the current Big Data era, Deep Neural Networks (DNNs) have quickly emerged as the new "killer app" for computing hardware. Data centers and cloud service providers are rapidly deploying DNNs to carry out a wide variety of tasks, such as speech recognition, natural language processing and computer vision. As the training and inference tasks become more complex, DNN workloads demand higher compute and memory capabilities. These trends have turbo-charged the growth of both special-purpose DNN accelerators [6, 18, 19, 24, 29, 35, 41, 44, 55, 55] and DNN-specific optimizations for general purpose CPUs [5].

While most research on DNN hardware has focused on scaling compute throughput (operations per second), the memory subsystem is quickly becoming a major bottleneck. The availability of higher resolution data and use of larger batch sizes has increased the size of input and intermediate datasets. More importantly, in order to cope with higher accuracy demands, data scientists have developed more complex and sophisticated DNN models with higher numbers of network layers and training parameters (e.g., 152 layers for Resnet, ILSVRC'15 winner [32] and 140 million parameters for VGG, ILSVRC'14 winner [51] vs. 8 layers for Alexnet, ILSVRC'12 winner [37]). These networks dynamically generate tens to hundreds of megabytes of intermediate data (activations or feature maps) per network layer, which needs to be buffered and communicated across multiple layers. This data often overwhelms the on-chip cache capacity and exerts immense pressure on main memory bandwidth. This has forced DNN hardware to use expensive memory solutions, such as HBM2 in Google's TPU2/3 [1].

In order to alleviate the bandwidth bottleneck of DNNs, prior works have observed that DNNs exhibit significant data sparsity, making them amenable to data compression techniques. In general, there are two types of data sparsity in DNNs: model sparsity and feature map sparsity. Model sparsity is often leveraged via offline techniques, such as model pruning and quantization. Since feature maps are generated dynamically and are unique for each input, it is harder to exploit feature map sparsity by using offline techniques. One promising technique to leverage feature map sparsity is to compress feature maps at runtime. Prior works on hardware DNN accelerators have proposed zero-skipping and zero value compression for cross-layer feature map communication [12, 19]. These techniques rely on fully-customized datapaths in order to encode or skip zero values in the activation data communicated between network layers. While these techniques are highly effective for custom-built accelerators, they are challenging for general purpose multi-processor CPUs that need to preserve legacy support for virtual memory and hardware-managed coherent cache hierarchies.

There is an intense ongoing debate regarding which hardware approach is most suitable for DNN training and inference: custom designed ASIC accelerators, FPGAs, GPUs or general-purpose CPUs. While accelerators and GPUs often have an edge over CPUs in terms of raw compute throughput, CPUs offer their unique sets of advantages as well. Recent studies at Facebook have shown that when tighter integration is desired between DNN and non-DNN tasks, CPUs are preferable due to their flexibility, lower latency, and availability in the datacenter [31]. Recent SIMD extensions, e.g., AVX512, have enhanced CPU performance for DNNs. Since CPUs are already widely deployed in data centers, client and edge devices, they provide a cost-effective vehicle for the growing DNN inference and training market, lowering the Total Cost of Ownership (TCO) compared with the design and development costs of accelerators. Therefore, it is important to develop CPU-tailored optimizations for addressing the memory challenges imposed by DNN workloads.

In this paper, we explore instruction set architecture (ISA) extensions for efficiently compressing dynamic data generated by DNNs. Our proposed techniques are motivated by the following two key observations: First, feature maps produced by DNN layers often include large populations of zero value activations interspersed with non-zero activations. Second, feature map accesses almost always exhibit perfect streaming patterns, whereby activation data is sequentially written by one layer and later read sequentially by another layer, without requiring random element retrievals. We propose ZCOMP, a CPU vector ISA extension tailored for DNN cross-layer communication. The function of ZCOMP is to dynamically compress/expand data in a memory region before being written/read to/from the memory hierarchy. It provides high-throughput parallel execution, transparent interactions with caches and virtual memory, and a flexible software interface. Specifically, ZCOMP enables zero data values in feature maps to be compactly encoded, while fully automating the meta-data generation, storage and retrieval, which eliminates the need for extra instruction executions and register usage.

Our paper makes the following key contributions:

- We evaluate feature map memory footprints, their sparsity, and impact on the memory subsystem for DNN workloads running on CPUs.
- We analyze existing AVX512-ISA approaches for feature map compression and demonstrate their limitations.
- We propose ZCOMP, an efficient vector ISA extension tailored for feature map compression, and explain its micro-architecture and software interface.
- We evaluate ZCOMP by detailed experimental studies using Intel Caffe ReLU activation layer for several DeepBench configurations and full networks from Google TensorFlow.
- Our results show that ZCOMP offers an average 31% reduction in memory traffic and an average 11% performance improvement for training popular DNNs.

2 BACKGROUND AND MOTIVATION

In this section, we provide a quick overview of DNNs. We also discuss the main memory needs of DNNs and the sparsity in cross-layer data transfers which motivates our work.

2.1 Deep Neural Networks

In recent years, Deep Neural Networks (DNNs) have been massively deployed by the computing industry to carry out various data mining , recognition and synthesis tasks. There are many types of DNNs, each suited for a specific set of applications. Multi-layer perceptrons (MLPs) were the first to use backpropagation for training and were applied to supervised learning problems. Recurrent Neural Networks (RNNs) such as Long-short-term-memory (LSTM) work well for tasks that exhibit time-dependent sequential behaviors, such as speech recognition and text-to-speech synthesis. Convolutional Neural Networks (CNNs) are most suitable for computer vision tasks, such as image recognition, image classification and object detection. In the rest of this section, we use CNNs as an example to explain the general structure and operation of DNNs.

A CNN consists of multiple network layers: an input layer, an output layer and many hidden layers. Each hidden layer is responsible for extracting certain features from input data and passing the extracted information to the next layer. Earlier hidden layers extract high-level features whereas later layers detect more detailed features that are eventually used to classify the image into one of the known categories. The most common layer types used in a CNN are as follows:

1. Convolution layers. CONV layers are the main building blocks of CNNs. Each CONV layer applies a set of filters to its input and generates a feature map which is passed on to the next layer. A filter in a CONV layer comprises a shared set of parameters (a.k.a. weights) which are applied to different portions of the input. The result of these convolutions is passed through an activation function, typically a Rectified Linear Unit (ReLU) to generate feature maps which are used as inputs for the next layer.

2. Pooling layers. These layers reduce the activation data size by combining information from multiple adjacent activations into a single activation, using either an averaging (AVG-POOL) or maximum (MAX-POOL) computation.

3. Fully-connected layers. FC layers connect every input to every output and are generally used as the last few layers of a CNN.

The data used in a CNN is classified into three categories:

1. Inputs. These are the input images that the network either uses to train itself or classifies/recognizes during inference tasks. The input data size depends on the image resolution and the batch size that is being processed by the CNN.

2. Weights. A CNN learns the weights for each network layer during the training process by comparing known outputs with current outputs and adjusting the weights accordingly. The total size of the weight data depends on the complexity of the network, such as number of layers and weight precision.

3. Activations/Feature maps. Each layer generates a set of output features by a product-sum of weights with inputs and then passing the results through a filter, such as ReLU. The input activations to a layer can either be the input images (in case of the first CONV layer) or activations propagated from the previous layer. The size of the activation data generated by a layer is a function of the input batch size and the number of filters used in the layer. Since pooling layers reduce the activation data size, latter layers in a CNN typically have smaller activation footprints than earlier layers.

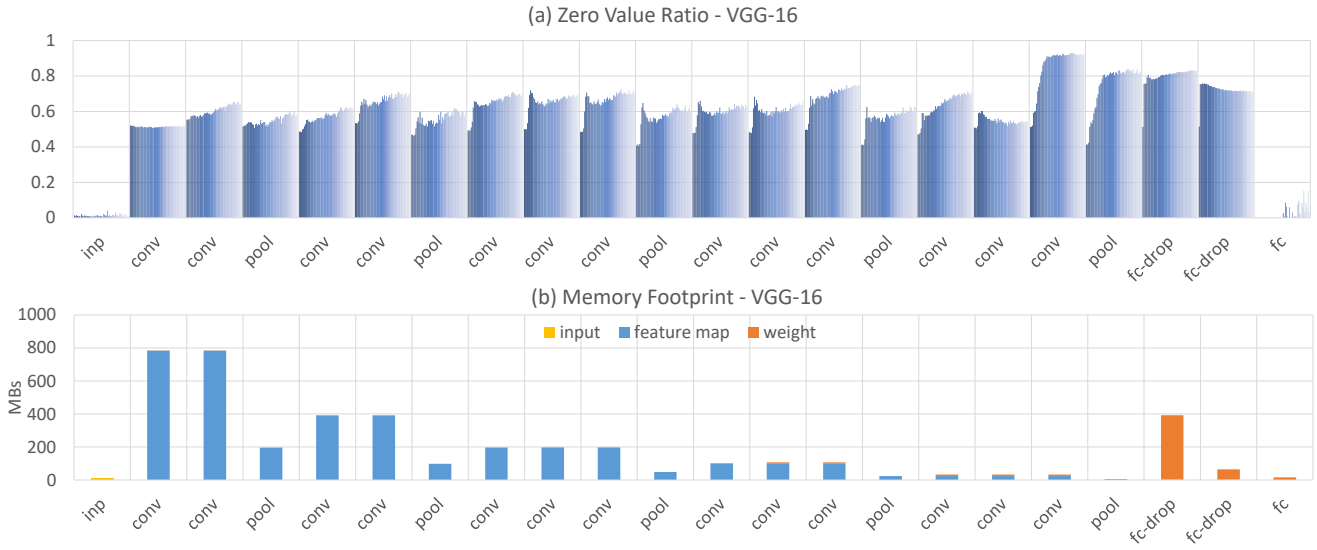

Figure 1: Zero value ratio and memory footprint for VGG-16 (batch size 64).

DNN training vs. inference. DNN training uses large amounts of input data (e.g., images), each with known outputs, in order to iteratively learn and update the weights in different network layers. The entire training process is broken up into a series of training epochs. Each epoch involves network traversals in both forward (called "forward propagation") and backward (called "backward propagation") directions. For ease of processing, the input data is broken up into smaller batches and each batch is propagated forward and backward once during an epoch. Network weights are updated during each training epoch. Training stops once the desired accuracy has been reached. The weights learned during training are then used to classify inputs during inference.

2.2 Sparsity in DNNs

There are two types of sparsity in DNNs: model sparsity and feature map (or activation map) sparsity. Model sparsity arises due to redundancy, zero values and over-precision of weights. Model sparsity is leveraged via pruning, quantization and re-training [29, 30]. Generally, compression techniques take trained sparse models as inputs and then analyze, quantize and compress them offline to achieve smaller models. These pruned models have fewer weights and often use reduced precision for inference, possibly preceded by re-training. Feature map compression, on the other hand, cannot be achieved with offline processing. Activation maps are generated uniquely for each input during both training and inference. Hence, the activation data needs to be dynamically compressed as it is being generated, which is more challenging than model compression.

Feature map sparsity arises due to two reasons. First, DNNs commonly employ rectified linear units (ReLU) as the activation functions. ReLU maps all negative input values to zeros in the output. Hence the activation maps generated from ReLU-based layers include large populations of zero values. Second, techniques such as drop-out randomly discard some of the output activations for algorithmic reasons (e.g. to avoid over-fitting). Discarded activations are represented as zero values in the feature map. As a result, feature maps exhibit a sparse behavior where large fractions of the elements are zeros. Since each network layer extracts a different set of features in input data, the location and number of zeros can vary significantly across different networks/layers/input combinations. Hence, a dynamic mechanism is needed to capture the sparsity in feature maps.

Some of the layers in the long DNN pipeline do not have activation functions at their outputs. For example, local response normalization (LRN) layers simply normalize the inputs to a range of values, or pooling layers selectively pass the inputs based on their values. Although these layers do not apply ReLU-based activation, they carry over the sparsity that is generated by the earlier layers.

Figure 1 demonstrates the feature map sparsity characteristics for VGG-16, one of the popular DNNs. For each network layer, we show the percentage of zero-value data generated in each training epoch. We carried out this analysis by profiling a TensorFlow implementation of VGG-16 with a batch size of 64, executed on a 16-core Skylake-X CPU. We observe that feature map sparsity exists at all network layers. Generally, pooling layers reduce the sparsity available at their inputs, whereas CONV layers mostly enhance it. In our analysis, we have merged the LRN layers with their previous layers and are not separately reporting sparsity for these layers. However, we note that LRN layers simply carry over the available sparsity from previous layers. Based on these insights, we conclude that in order to fully utilize the available feature map sparsity, a dynamic compression mechanism needs to be generalized to apply to all cross-layer data transfers and not just to the activation maps generated by ReLU functions.

2.3 Memory Requirements for DNNs

Although most prior work on DNNs has focused on their compute bottlenecks, DNNs also spend significant amount of time on data movements throughout the memory hierarchy. To quantify the memory impact on DNN performance, we profiled 5 popular DNNs implemented using the TensorFlow framework on the Sniper simulator modeling an AVX512-capable 16-core CPU (details about

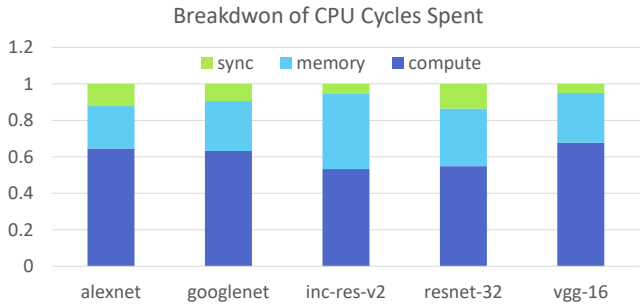

Figure 2: CPU cycle breakdown for DNN benchmarks.

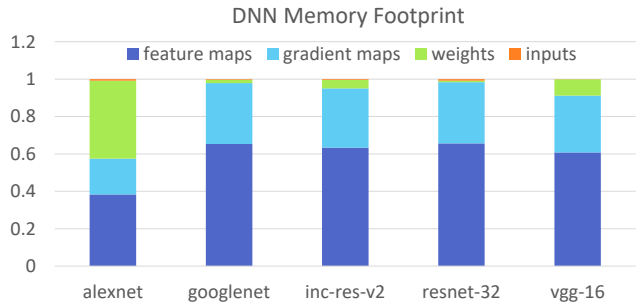

Figure 3: Memory footprint of key data structures for DNN benchmarks.

our methodology are provided in Section 5). Figure 2 shows the breakdown of CPU stall cycles caused by compute, memory and synchronization operations. It is worth noting that DNN workloads spend a large fraction (24%-41%) of their execution time stalled due to memory accesses. This data motivates the need for memory optimizations tailored for DNNs.

The memory system bottleneck will become more critical in the future due to two key reasons. First, processors will continue to incorporate hardware customizations and ISA extensions to improve raw compute throughput, which would reduce compute stall cycles. Second, future trends in DNNs are likely to put more pressure on on-chip cache capacity and memory bandwidth [42]. For example, more accurate implementations of common tasks, such as image understanding, object detection, action recognition and video inference require larger feature maps. Furthermore, for distributed large-scale inference, larger batches are preferred to improve throughput which increases the memory bandwidth and capacity needs.

We analyzed the memory capacity usage of DNN benchmarks. Figure 3 shows the memory capacity consumed by different data types in each DNN. For this analysis, we used a batch size of 64 for all networks except for ResNet, which uses 128. We observe that cross-layer feature map data accounts for the majority of the memory footprint. Another big source of memory usage is the gradient maps which are communicated between layers during backward propagation. We also carried out a layer-by-layer analysis of the feature map and weight footprints for the VGG-16 network, shown in Figure 1(b). We note that earlier DNN layers, which are generally very wide, often generate hundreds of MegaBytes of cross-layer feature map data, whereas the weight data is only dominant in the fully-connected layers. Moreover, the use of larger batch sizes will cause further increases in the feature map footprint relative to the weight footprint.

It is also worth noting that feature map data generated by a layer exhibits distinct short-term and long-term reuses. First, feature maps from one layer are passed to the next layer as inputs. Due to the all-to-all nature of inter-layer communication, all the activation data in a layer is fully calculated, and then passed to the next layer. This short-term reuse often stresses only the on-chip cache capacity and bandwidth. However, for complex networks and large batch sizes, the feature map produced by even a single layer can exceed the on-chip cache capacity, thereby exposing the main memory bandwidth bottleneck. Second, feature maps that were calculated during the forward pass are re-used during the backward pass in order to calculate the weight changes. Since feature maps generated by all layers need to be accumulated and remain buffered in the main memory during the forward pass, this long-term reuse exerts pressure on both main memory capacity and bandwidth.

2.4 CPUs for Deep Learning

Substantial performance improvements for deep learning workloads are achieved by using various accelerators including GPUs [6, 20], FPGAs [24, 40], and custom ASIC systems [19, 35, 41, 55]. However, CPUs still provide unique advantages for deep learning workloads. Recent studies at Facebook ([31, 42]) highlight that availability, flexibility and low latency of CPU platforms are highly valuable for deep learning. CPUs are preferred especially when a tighter integration between deep learning workloads with other business services is needed. CPUs also provide cost-effectiveness due to the additional design and deployment costs of accelerators. Consequently, inference workloads are run primarily on CPUs, whereas training often targets CPUs, GPUs or both for various services [31].

Recent SIMD ISA extensions, e.g., AVX512, have enhanced CPU performance in data-parallel computing, making CPUs more attractive for DNNs. Several new Vector Neural Network Instructions (VNNI) are included in AVX512 extensions, which specifically target DNN workload requirements and provide hardware customizations to the CPUs in the form of ISA extensions. As these instructions improve CPUs compute capability for deep learning, the performance impact of memory becomes even more important.

3 ZCOMP MICRO-ARCHITECTURE

In this section, we present `ZCOMP` SIMD instruction family as a vector extension that is similar to other instruction families within AVX512 of x86 ISA [5]. We define two new `ZCOMP` instructions: `zcompl` and `zcomps`. `zcompl` is used to load and decompress the elements, whereas `zcomps` compresses and stores them. As is common in x86, each instruction has multiple variants to support different data types (e.g. int8, fp16, int, fp32, double, etc.). Throughout the paper, we will assume 32-bit float by default, as it is a common data type implemented in the deep learning frameworks we evaluated (Tensorflow [7] and Caffe [34]) and fp16 support is limited in current CPUs. However, the proposed concepts can be easily extended to other data types. We next describe two different variants of `zcompl` and `zcomps`: (i) interleaved header and (ii) separate header.

3.1 Interleaved Header `zcomps` and `zcompl`

`zcomps` is a SIMD vector instruction that operates on 512 bit data similar to other x86-AVX512 instructions (but it can be generalized to target different vector widths). The micro-architecture of `zcomps` with interleaved header is shown in Figure 4. Interleaved-header `zcomps` requires two input registers (reg1 and reg2) and a comparison condition flag (CCF). It is used in the form of "`zcomps` [reg2], reg1, #CCF". It takes the input vector via the reg1 register operand (e.g. a 512-bit SIMD register) and stores the compressed version into the memory region starting from the location pointed by the reg2 operand (i.e. register indirect access as represented with [ ]). Finally, the third operand, #CCF, is an immediate field that holds the comparison condition for the elements in the input vector. The comparison flag can be configured to simply check if the elements are zero valued and compress them in the cross-layer data. More interestingly, it can be configured to check if the elements are less than or equal to zero values in the input vector. This can be very useful in ReLU activation layer to fuse (i) the activation function, which essentially performs a less than or equal to comparison, and (ii) the compression into a single instruction in the form of `zcomps`.

`zcomps` execution starts with performing the element-wise comparison operation of the vector in reg1 based on the comparison condition flag CCF. The result of the CCF vector comparison creates a mask with 1 bit per element that will represent the compression metadata (i.e. header in Figure 4). Meanwhile, the number of uncompressed values in the vector (which can be non-zero values or non-negative numbers based on the CCF) is calculated by simply counting the 1s in the comparison result. Then, only the elements in the vector lanes which are uncompressed are selected. The final output to be written into memory is formed by concatenating the header and the selected uncompressed elements sequentially. This final output is written into the memory region starting from the address held by reg2. After the execution, reg1 remains unchanged and reg2 is automatically incremented to account for the amount of data plus metadata that is written into the memory after compression. When `zcomps` is used iteratively within a loop to compress larger data arrays, reg2 acts as a compressed data pointer and the auto-increment feature enables the next iteration to fill in the new elements starting from the location that the current iteration left off.

In the example demonstrated in Figure 4, there are 6 non-zero 32-bit elements out of 16-element 512-bit vector. In this example, we assume that CCF is configured to check if the elements are zero-valued. Based on the lanes of non-zero elements, the result of the vector comparison becomes 1001000100011100. This bit vector constitutes the compression metadata, i.e. header (0x911C). Based on this bit vector, active input lanes are selected and shifted to gather the non-zero elements together. The header and the non-zero elements are then concatenated to form the final output to be written into the memory region starting from 0x1000, the address pointed by reg2. In addition, since reg2 acts as a compressed data pointer, it is automatically incremented to account for the total data written out, including both the header and the compressed data. In order to calculate the size of compressed data, we perform a population count (popcount) operation on the comparison bit vector. The popcount output is the number of non-zero elements (NNZ), 6 in

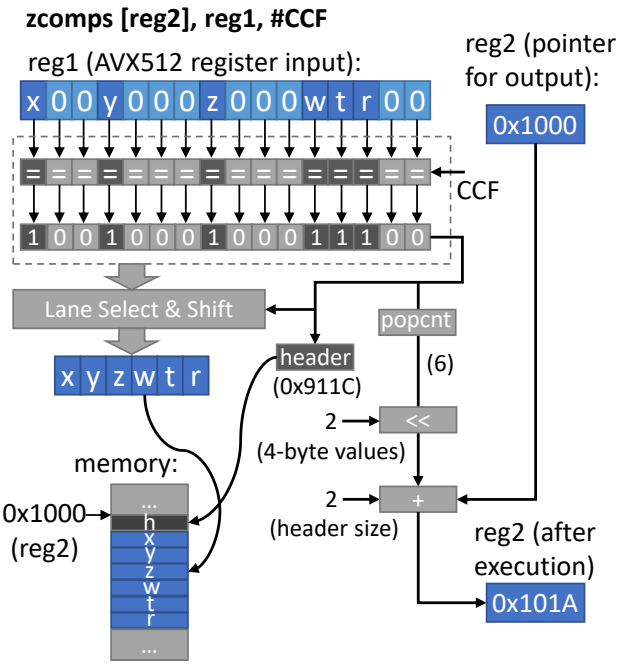

Figure 4: Micro-architecture for `zcomps` instruction with interleaved header (512-bit vector fp32 elements).

the example of Figure 4. Since we are dealing with 4-byte elements (fp32), total amount of data written out is determined as 6*4-bytes (left-shift NNZ by 2) for the compressed non-zero elements and another 2-bytes (increment by 2) for the 16-bit header, which makes a total of 26 bytes. Assuming a byte-addressable memory space, contents of reg2 is incremented by this amount and it becomes 0x101A where the initial content was 0x1000. If `zcomps` is used in a loop iteratively, the next set of elements will be compressed into the memory starting from the address 0x101A, and this operation will continue.

Before going into other `zcomps` implementation details regarding memory allocation, virtual memory, and data alignment, we first explain `zcompl` instruction that performs decompression. `zcompl` is a dual of `zcomps`; any data compressed and stored via `zcomps` has to be retrieved by using `zcompl`. Similar to `zcomps`, `zcompl` takes two input registers: reg1 and reg2, but it does not require an immediate CCF field. It is used in the form: "`zcompl` reg1, [reg2]". The first operand, reg1, is the destination register (e.g. 512-bit SIMD register) to which the loaded vector will be expanded. The second operand, reg2, is the source pointer for the loaded vector. Note that while the vector size after expansion is known, the source data can have any arbitrary size based on its compression ratio. However, the information regarding the source data size is already contained in the header.

`zcompl` first reads the bytes that correspond to the header starting from the address held by reg2. Note that while the header size depends on the element precision and vector width, it is nevertheless a known value. For example, for 512-bit vector with 32-bit elements, the mask will be 16 bits. A bit-wise comparison between the header and a zero bit-vector determines the active lanes that

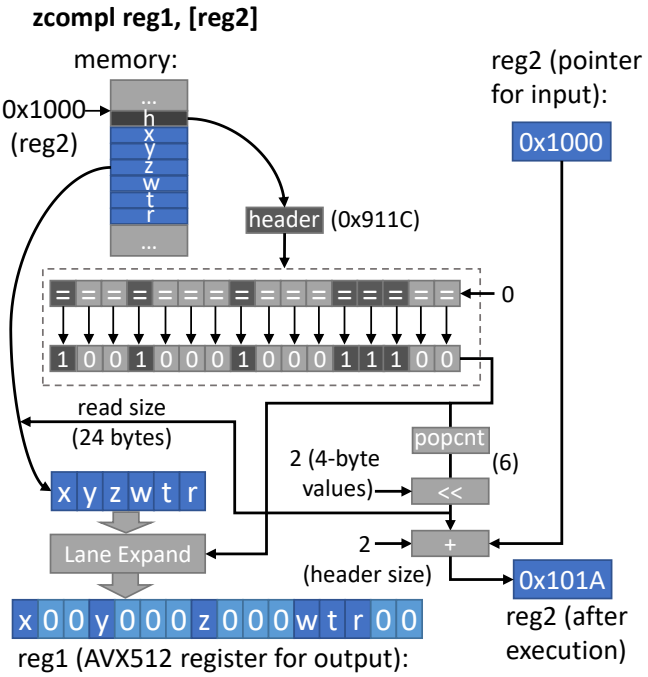

Figure 5: Micro-architecture for `zcompl` instruction with interleaved header (512-bit vector fp32 elements).

will have non-zero values. A following popcount operation on the comparison result also determines the number of non-zero (NNZ) elements to be read from the memory. Consequently, `zcompl` reads in NNZ elements starting from the address held by reg2 plus the header size. These elements are then expanded into a full reg1 register based on the active lanes determined by the zero bit-vector comparison. Inactive lanes, i.e. lanes that have 0 bits in the corresponding comparison result, get a 0 value element in the expanded vector in reg1. After the execution, reg1 will hold the expanded vector and reg2 is automatically incremented to account for the amount of compressed data plus metadata that is read from memory. Similar to the `zcomps`, if `zcompl` is used iteratively within a loop, reg2 will point to the address of next vector to be read.

3.2 Separate Header `zcomps` and `zcompl`

The interleaved header approach aims to fit headers and the compressed data into the original memory space. For large datasets with certain sparsity guarantees, that approach can be very efficient. However, if the compressibility of the data is completely unknown, separately and explicitly allocating memory for the metadata is desirable. To achieve this goal, the second variation of `ZCOMP` instructions, separate-header, decouples the metadata (header) and the compressed data storage and retrieval.

The ISA for separate-header `zcomps` and `zcompl` is very similar to the interleaved-header variants. But instead of two input register operands, we need three inputs for separate-header instructions, used in the form of "`zcomps` [reg2], reg1, [reg3], #CCF" and "`zcompl` reg1, [reg2], [reg3]." The additional register reg3 holds the pointer to the separated header store. We skip the detailed micro-architecture descriptions as they are very similar to the interleaved-header variants. For `zcomps`, the execution only differs at the final stage where the generated header is written into the memory location pointed by reg3 instead of concatenating it in front of the compressed vector. Similar to reg2, reg3 is also auto incremented to enable a continuous iterative operation. For `zcompl`, execution starts with reading in the header from the location pointed by reg3 instead of reading it from reg2 pointer. Note that the reg2 auto-increment amount is adjusted to account only for the compressed data size.

3.3 Micro-architecture Details

Execution Pipeline. Both `zcomps` and `zcompl` instructions include several operations as part of their execution. These operations can be divided into two categories: logic micro-ops and memory micro-ops.

We provide a detailed critical path overview for the logic ops as follows: The logic component is divided into two pipeline cycles based on the corresponding micro-op latencies reported for modern CPUs [23]. For `zcomps`, execution starts with a vector comparison. For the feature-map compression usage, only "equal to zero" and "less then or equal to zero" operations are required. Assuming fp32 elements, this is achieved via 1-bit XOR (for sign) and a 32-bit OR (for equal to zero) operation per lane which can be implemented via 6x2-input gates or 4x3-input gates per lane. This operation is followed with a popcount on the 32-bit binary vector which is achieved with a hierarchical carry look-ahead adder tree. The 5-bit result of popcount is padded by two zeros (i.e., equivalent of left-shift by 2 since we have 4 bytes per element). In parallel to the popcount, elements from the active lanes are selected and concatenated together. This constitutes the end of the first pipeline cycle. In the second cycle, a tree of carry look-ahead adder adds the 7-bit popcount (with padded zeros), the 2-bit header size and the 48-bit virtual address held in reg2 together.

Although the logic micro-op execution has a 2-cycle latency, because of the pipelined implementation, it achieves a 1 instruction per cycle throughput (i.e. it can accept a new instruction every cycle). Memory micro-ops have variable latency based on the location of the data (L1/L2/LLC/DRAM) and the size of the transfer which will be added on top of the logic latency. Nevertheless, due to prefetching, parallel execution (explained in Section 4), and bulk transfer of large feature maps, `ZCOMP` usage becomes throughput-bound which effectively hides the latency of the logic component. In fact, when we test a 3-cycle logic latency variant, the overall performance is almost identical to the 2-cycle version due to throughput-bound operation.

Prefetching. Due to the sequential nature of `ZCOMP`-based compression/expansion, memory latency can be an issue. Especially for expansion, reading the header, compressed data, then the next header and the next compressed data are all sequentially dependent. We observe that, due to sequential memory layout and access patterns, both compressed data and headers are easily prefetchable with a streaming prefetcher. `ZCOMP` generated memory micro-ops train the L2 streaming prefetcher and trigger subsequent prefetches. Consequently, while a core is working on the current vector, the

memory requests for the next vector(s) will be in flight, which improves the memory-level parallelism and mitigates the memory latency. For the workloads analyzed in Section 5, we observe L2 prefetcher accuracy of 98-99% and coverage of 94-97%.

Alignment. Note that, even if the available sparsity enables the compressed data and the headers to fit within original memory allocation boundaries, there can be individual vectors in the dataset which may not be compressed. For the targeted 512-bit SIMD operation, this leads to header plus data to occupy more than a single 64-byte cacheline. These cases are handled the same way as a regular unaligned store instruction that falls off a cacheline boundary. Moreover, based on the data-type precision, compressed vectors can end up in various sizes. 4-byte elements with 2-byte headers and 2-byte elements with 4-byte headers both guarantee 2-byte aligned memory transfers for the compressed vector which is already supported by the existing micro-architectures. But for lower precisions, alignment issues may lead to redundant data transfers.

4 ZCOMP USAGE IN DNN PROCESSING

4.1 Virtual Memory Allocation Impacts

ZCOMP provides a software-friendly interface for drop-in replacement of store and load instructions with zcomps and zcompl for DNN cross-layer data transfers. With ZCOMP, the software does not need to worry about managing compression metadata or checking if the data is actually compressible. However, another important aspect is the memory allocation and virtual memory interaction for compressed data, which needs to be programmer-friendly in order to be effective.

The feature map compression ratios are dynamic parameters that depend on the inputs. Therefore, it is not possible to allocate the memory space before passing the inputs through the layers. Even if the virtual memory allocation is somehow done based on the compressed size estimations, it needs to account for the worst case compression, unless the intermediate buffers are allocated/deallocated for every input batch, which can lead to performance issues.

ZCOMP instructions rely on the original virtual memory allocations and do not aim to reduce virtual memory footprint. Instead, our goal is to reduce the actual physical memory space usage without changing the original virtual memory allocations, if possible. Interleaved header and separate header variants of ZCOMP instructions (Section 3) interact differently with the virtual memory. If the estimated compression ratio is large enough to make the metadata and the compressed data to fit in the originally allocated virtual memory space, then interleaved header ZCOMP instructions are preferred. For example, considering 512-bit SIMD instructions and fp32 values, only an overall 3.125% compressibility is sufficient to fully amortize the metadata. In this case, the original memory allocations can remain unchanged, whereby the metadata will be interleaved with the compressed data without exceeding the original allocation space. Compressed data pointers will always remain within the allocated address space. However, a memory violation can happen without enough compressibility.

If the compressibility is completely unknown, there are two different options for memory allocation. First option is to continue to use interleaved header, but to modify the memory allocations for the intermediate data buffers to account for the full uncompressed

```
• void _mm512_zcomps_i_ps(__m512* &out_ptr, __m512 inp_vec,
                          uint8 CCF);
• __m512 _mm512_zcompl_i_ps(__m512* &inp_ptr);
• void _mm512_zcomps_s_ps(__m512* &out_ptr, __m512 inp_vec,
                          __mmask16* &headers, uint8 CCF);
• __m512 _mm512_zcompl_s_ps(__m512* &inp_ptr,
                          __mmask16* &headers);
```

Figure 6: ZCOMP software interface using intrinsics.

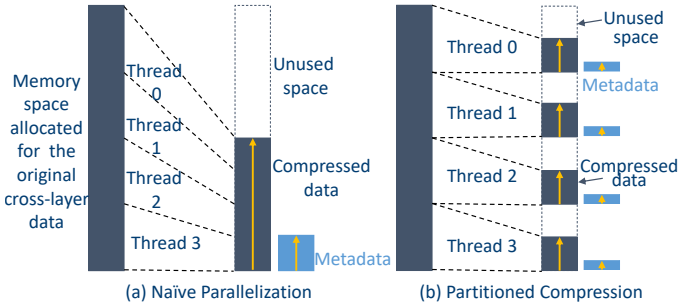

Figure 7: ZCOMP parallelization approaches.

data plus the metadata size. Second option is to use separate-header variant where the metadata is completely decoupled from the data. This requires an additional memory allocation for the header store, however, the original allocation for the intermediate data buffers remains unchanged. These options ensure that the compressed data pointers will always stay within the allocated address space, eliminating the memory violation possibility.

In each of these techniques, irrespective of whether additional virtual memory is allocated or not, the usage of physical memory space will be reduced based on the compression ratio. Reducing physical memory footprint decreases the bandwidth and capacity usage throughout the cache hierarchy and DRAM. For the DNN benchmarks profiled, we observe that average sparsities are within the range of 49%–62%. Hence, throughout our analysis, we have used the interleaved header and kept the original allocations unchanged. Since the compression ratios are more than sufficient to amortize the metadata, ZCOMP is able to seamlessly fit headers and compressed data within the original allocation boundaries by using a much smaller physical memory footprint.

4.2 Software API via Intrinsic Functions

The most effective way to apply ZCOMP is to use a corresponding intrinsic function to emit zcomps and zcompl instructions. Intrinsic functions are a common way of interfacing to similar SIMD ISA extensions such as AVX512 [3]. Proposed intrinsic functions for zcomps and zcompl are given in Figure 6, which demonstrates the case for interleaved (i) and separate (s) header with 512-bit vector fp32 elements. Note that input and output pointers use a pass-by-reference construct to allow them to be auto-incremented to point to the next vector (these pointers will be held in reg2 operand). For separate-header, header pointer also uses this apporach to enable auto-increment (reg3).

Proposed functions provide a simple software interface such that a single intrinsic function for compress/expand simply replaces the vector store/load operation. The programmer does not need to worry about generating masks to perform masked load/store, handling metadata and maintaining pointers to where the compressed

data is. However, we trade away the capability to access elements randomly. Both compression and expansion needs to be performed vector-by-vector sequentially for each slice of the dataset. As DNN intermediate data buffers are communicated between layers or re-read in the backward pass in bulk, this is a reasonable trade-off.

Reading and writing the intermediate data buffers in DNNs can be clearly identified in the beginning and at the end of different types of layers, where the proposed intrinsic functions are inserted. These insertions require minor changes in DNN software. Nevertheless, we note that these changes can easily be incorporated in the Deep Learning frameworks and would not burden the application developers, for whom these optimizations would enable a transparent performance improvement.

4.3 Parallel Execution

So far, we have focused on the case where the compression/expansion using ZCOMP is performed sequentially, vector by vector. As the sizes of the compressed vectors are embedded in the headers, operating on the next vector requires the completion of the current one. Hence, ZCOMP throughput can be an issue without parallel execution support.

For a naive parallelization approach which aims to compress the feature map into a contiguous chunk, a heavy serialization occurs to pass the compressed data pointer between threads (see Figure 7(a)). Instead, slicing the feature map into chunks where each thread will compress their own portion as an independent stream can be a better parallelization strategy. For this partitioned compression case (Figure 7(b)), each thread gets the same amount of memory region to work on, but they will have separate compressed data pointers which are isolated from each other. Elements within a chunk still need to be compressed sequentially but each chunk can be compressed in parallel, independently. With enough chunks that can sustain the available cache/memory bandwidth, the throughput problem can be mitigated. Note that the expansion needs to match the compression parallelization strategy to correctly retrieve the data. Similarly for the expansion, each chunk is expanded separately in parallel where the expansion is still sequential within a chunk.

In the iterative usage of ZCOMP instructions, compiler optimizations such as loop unrolling become very challenging due to loop-carried dependencies. Partitioned compression technique can be further extended to address this problem. If the chunk assigned to a thread is further sliced into smaller sub-blocks, then, at every iteration, multiple ZCOMP instructions can be executed that target different sub-blocks. The number of sub-blocks determines the degree of unrolling that can be performed; however, large number of chunks/sub-blocks can lead to fragmentation. Overall, we observe that loop unrolling has minor impact for large feature-maps. For small feature map sizes, it can improve instruction throughput; therefore we match the ZCOMP unrolling to the default unrolling performed by the compiler for the baseline case. Loop tiling/reordering changes the traversal order from the baseline data layout. Hence, expansion needs to be done considering the loop traversal for compression. If compressed data needs to be read in a different order, an explicit data reorganization should be performed.

```
// Parallel zcomps ReLU Activation Layer
#pragma omp threadprivate(Y_ptr)
#pragma omp parallel
{
  // Each thread will get a separate slice of Y
  Y_ptr = Y + threadID*n/num_threads;
}
#pragma omp parallel for
for (int i=0; i<n/vec_elems; i++)
{
  __m512 tvec = _mm512_load_ps(X+i*vec_elems);
  _mm512_zcomps_i_ps(Y_ptr, tvec, _LTEZ);
}
```

Figure 8: Storing ReLU generated feature maps by using the proposed `zcomps` instruction.

```
// Parallel zcompl to Retrieve Data After Compression
#pragma omp threadprivate(X_ptr)
#pragma omp parallel
{
  // Each thread will get a separate slice of X
  X_ptr = X + threadID*n/num_threads;
}
#pragma omp parallel for
for (int i=0; i<n/vec_elems; i++)
{
  __m512 tvec = _mm512_zcompl_i_ps(X_ptr);
  // Use the retrieved input tvec …
}
```

Figure 9: Retrieving cross-layer feature maps by using the proposed `zcompl` instruction.

4.4 AVX512 and ZCOMP Usage Comparison

We compare the usage efficiency of ZCOMP with pre-existing compression support in AVX ISA. For this purpose, we present code examples which store and retrieve sparse feature maps by using intrinsic functions. In each of the examples, parallelization is expressed via OpenMP pragma directives.

We first start by tackling a ReLU activation layer. We omit a straightforward parallel vectorized baseline implementation that uses AVX512 store/load and vector comparison to determine the activation results. Figure 8 demonstrates the case for using interleaved header `zcomps` via partitioned compression parallelization. We assume a ReLU activation over n element input array X to be written into Y. First, the target memory space is sliced into chunks where each thread gets a different compressed data pointer (Y_ptr). In this example, `zcomps` performs a "less then or equal to zero ($\_LTEZ$)" comparison to map all non-positive numbers to zero and compress the results in one-shot. Note that, for a generic layer that does not perform ReLU comparison, condition flag can be set to $\_EQZ$ (equal to zero) and `zcomps` can be used similarly to write out sparse feature maps in the compressed form.

When the feature maps are written out using `zcomps`, they need to be retrieved back using `zcompl`. Figure 9 demonstrates the use case for `zcompl` to retrieve back and expand the compressed data. Similar to the previous case, each thread will get a separate slice to expand (X_{ptr}). In this case, we do not have a comparison flag as `zcompl` expands the non-zero values to their lanes embedded in the header while only inserting zeros for the other lanes in the vector.

Next, we present applying recent AVX512 vcompress and vexpand instructions for the above use cases [5]. Since these are more generic instructions, applying them to the previous examples consumes several additional instructions and registers. Figure 10 and

```
// Parallel AVX512-Vcompress ReLU Activation Layer
__m512 zvec = _mm512_setzero_ps();
#pragma omp threadprivate(index)
#pragma omp parallel
{
  // each thread will get a separate slice of Y
  index = threadID*n/num_threads;
}
#pragma omp parallel for
for (int i=0; i<n/vec_elems; i++)
{
  __m512 tvec = _mm512_load_ps(X+i*vec_elems);
  __mmask16 mask = _mm512_cmp_ps_mask(tvec, zvec, _NEQ);
  uint nnz_cnt = _mm_popcnt_u32(mask);
  _mm512_mask_compressstoreu_ps(Y+index, mask, tvec);
  index += nnz_cnt;
  headers[i] = mask;
}
```

Figure 10: Storing ReLU generated feature maps by using AVX512 compress instruction.

```
// Parallel AVX512-Vexpand Retrieve Data After Compression
#pragma omp threadprivate(index)
#pragma omp parallel
{
  // Each thread will get a separate slice of X
  index = threadID*n/num_threads;
}
#pragma omp parallel for
for (int i=0; i<n/vec_elems; i++)
{
  __mmask16 mask = headers[i];
  __m512 tvec = _mm512_maskz_expandload_ps(mask, X+index);
  uint nnz_cnt = _mm_popcnt_u32(mask);
  index += nnz_cnt;
  // Use the retrieved input tvec …
}
```

Figure 11: Retrieving cross-layer feature maps by using AVX512 expand instruction.

Figure 11 show the usage of AVX512 vcompress and vexpand instructions for the ReLU activation layer and compressed data retrieval, respectively. Both examples use the partitioned compression parallelization similar to the ZCOMP use. We omit the details for the additional intrinsic functions used, which can be found in [3]. Upon disassembling the code snippets in Figures 8–11, we observe that AVX512 vcompress and vexpand require 5-6 extra static scalar/vector instructions inside the loop body, and use 4-5 additional registers, compared to ZCOMP. Hence, over many iterations, AVX512 implementations incur larger dynamic instruction overheads, which we quantify in detail in Section 5.

5 EVALUATIONS

5.1 Simulation Methodology

Our evaluations are based on the Sniper multi-core simulator [15]. We use an extended in-house version of the Sniper simulator that includes a detailed micro-architecture model for the recent ISA extensions, such as AVX512. Our target architecture details are summarized in Table 1. We have included ZCOMP micro-architecture support in Sniper that models the necessary memory accesses and micro-ops required by ZCOMP instructions. Since existing compilers cannot generate ZCOMP instructions, our methodology is to use AVX512 store/load instructions instead of `zcomps/zcompl` but label them differently. In the simulator, when a labelled AVX512 store/load is encountered, the default behavior is overridden and the instructions are treated as `zcomps/zcompl`.

Table 1: Architecture Configuration

Core	16 cores, x86 AVX512, 2.4 GHz, 4-issue
L1-D/I	32 KB private, 8-way, LRU
L2	1 MB private, 16-way, SRRIP
L3	24 MB shared, 12-way, SRRIP
Prefetcher	Stream/stride at L2, IP-based at L1
NoC	2D-mesh, XY routing, 2-cycle hop
Memory	4 channels, DDR4-2133, total 68 GB/s BW

5.2 ReLU Activation Layer

We start our evaluation with the use case of applying AVX512 compress/expand instructions and our proposed ZCOMP instructions to the ReLU-based activation layers. We take the ReLU layer implementation from Intel optimized Caffe DL framework [34], which is vectorized to use AVX512 instructions. ReLU layer typically follows convolutional and fully-connected layers in DNNs. As the inputs to the ReLU layer, we use various tensor shapes from the Baidu DeepBench benchmark suite [2] (x-axis of Figure 12). We use a total of 44 inputs collected from training and inference-server suites for convolutional and fully-connected layers. Although default DeepBench benchmarks use a random initialization of the values, we use uncompressed snapshots from the evaluated DNN feature maps (with an average 53% sparsity) to initialize the input tensors in order to accurately capture the effects of compression.

In addition to the baseline AVX512 vectorized version (avx512-vec), we implemented a version that uses AVX512 vcompress/vexpand instructions as presented in Figure 10 to perform a compression-enabled ReLU activation (avx512-comp). Finally, we implemented a ZCOMP version using the approach presented in Figure 8 and following the methodology described in 5.1. Our evaluation results are presented in Figures 12(a), (b) and (c). Each benchmark group is sorted according to their input tensor sizes ranging from only few KBs up to 560 MBs.

Figure 12(a) presents the data traffic between the cores and the cache hierarchy. We observe that avx512-comp and ZCOMP both capture the available sparsity and reduce the data traffic by 42% and 46%, on average. Moreover, Figure 12(b) presents the data movements to/from off-chip DRAM. For smaller feature map sizes, we do not observe substantial DRAM traffic, except for the compulsory accesses. But, for larger sizes, avx512-comp and ZCOMP both provide significant reductions, by 48% and 54%, on average. Explicit management of the masks, storage/retrieval from separate memory locations, and executing more dynamic instructions lead to higher data traffic for avx512-comp compared to ZCOMP. In Figure 12(b), we observe a big drop in traffic for certain inputs within each benchmark group (except conv-infer) that suggests a sweet spot for off-chip traffic reduction. This sudden drop happens when the original feature map would not fit in the on-chip caches but compression makes it fit and therefore heavily reduces off-chip DRAM accesses. Inference benchmarks generally have much smaller batch sizes and feature maps. Therefore, for the conv-infer benchmark group, feature maps of a single layer almost always fit in caches and we do not see a big drop in off-chip accesses.

For avx512-comp, that uses existing AVX512 instructions, several extra operations induce a latency overhead and decrease the instruction throughput (Section 4). For larger feature map sizes,

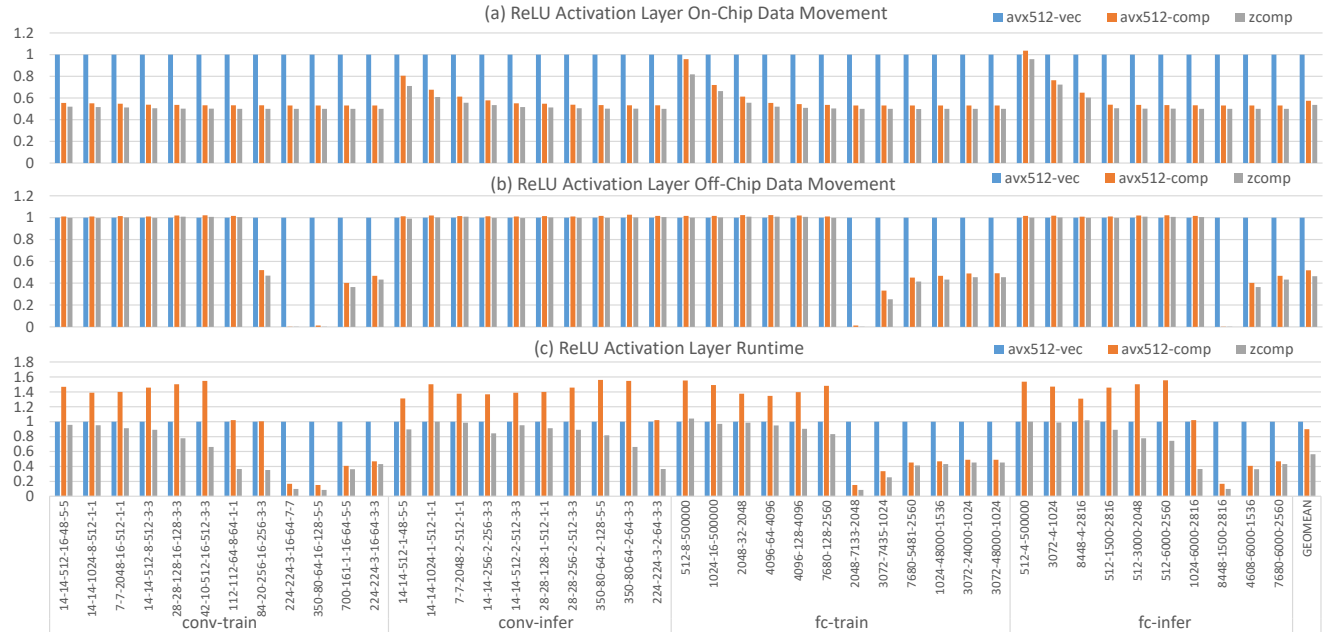

Figure 12: ZCOMP benefits over existing AVX512 instructions for ReLU activation layers with DeepBench input shapes.

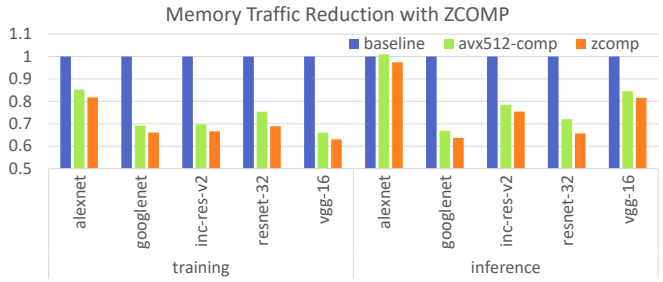

Figure 13: ZCOMP data traffic reduction for full networks.

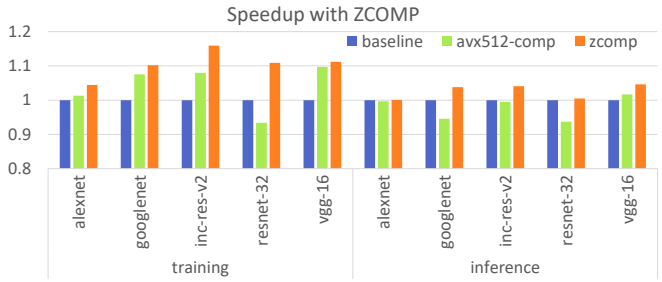

Figure 14: ZCOMP speed-up for full networks.

avx512-comp can be a viable approach where the savings due to reduced data movements could amortize the instruction overheads (Figure 12(c)). However, for small/medium feature maps. this approach can lead to severe performance degradation, as shown in Figure 12(c). Although a runtime mechanism that selectively applies AVX512 compression can be a viable approach, it requires knowledge about the hardware platform to estimate if the feature maps will be cache/memory resident. Moreover, for end-to-end networks, simply looking at the current layer's feature map size is not sufficient as there are other data structures and prior layer's maps that dynamically share memory.

In contrast to avx512-comp, ZCOMP provides a performance upside for both small and large feature maps. On average, ZCOMP achieves a performance improvement of 77% over the avx512-vec and 56% over avx512-comp. There are only 2 outliers out of 44 cases, where ZCOMP does worse than avx512-vec, by only 2% and 4%. For these cases, the performance headroom does not amortize the overheads since small input datasets easily fit in the L1 cache. In Figure 12(c), we again observe the big drop in the runtime, correlated with the reduced off-chip data traffic. In these cases, ZCOMP achieves super-linear speed-ups of up to 12x.

5.3 Full Networks

Our evaluations for full networks are based on 5 DNNs (AlexNet, GoogLeNet, Inception-Resnet-v2, Resnet-32, VGG-16) implemented within TensorFlow framework configured to use Intel MKL (Math Kernel Library) [4]. For training, we used Oxford flowers (1360 images) and a 100,000 image subset of the Imagenet dataset. For these DNNs, we observe that memory based stall cycles account for 24-41% of the overall runtime. Moreover, majority of the memory accesses originate from cross-layer feature map transfers, which account for 40-65% of the data traffic. Feature maps monitored from the evaluated networks show 49-63% average sparsity (overall 53%), which creates a great opportunity to apply ZCOMP.

Our full network evaluation results are shown in Figure 13 and Figure 14. For training benchmarks, we use a batch size of 64 for all the networks (except for ResNet which uses 128). For inference, generally smaller batches are preferred to reduce the latency; therefore we use a batch size of 4 for all benchmarks. Due to large batch sizes, data movements in training are dominated by feature maps. In contrast, in inference, weight transfers also become a major factor. However, due to the lack of backward gradient maps in inference, feature maps still account for a large fraction of the allocated

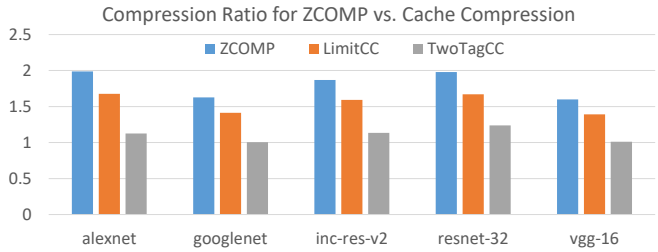

Figure 15: ZCOMP vs. cache compression.

memory (44% on average). As shown in Figure 13, average data traffic reductions for training/inference are 31%/23% for ZCOMP and 26%/19% for avx512-comp.

In inference, generated feature maps are discarded as soon as they are used by the next layer. Therefore, the data movement savings can target only short term reuse on small data transfers. For training, on the other hand, feature maps accumulate layer by layer, leading to much larger active data, and a potential for larger performance improvements. Consequently, performance improvements of using ZCOMP are on average 11% and 3% (up to 16% and 5%) for training and inference, respectively, as shown in Figure 14. In comparison, existing AVX512 compression results in slowdowns for 5 out of 10 benchmarks due to the following overheads: (i) additional instructions, (ii) register usage, and (iii) metadata management. On average, avx512-comp achieves 4% speed-up for training but 2% slowdown for inference. We conclude that ZCOMP is a reliable approach that provides substantial improvements for memory-intensive layers but has little to no overhead when the opportunity is limited.

5.4 Comparison with Cache Compression

Cache compression has been proposed to increase effective cache capacity and reduce memory traffic at the cost of higher cache hit latencies [10, 14, 17, 22, 28, 43, 47, 48]. We compare the effective compression ratio for ZCOMP to that of cache compression. We analyzed the compression ratio of five random static snapshots from the five DNN workloads. For these snapshots, average compression ratios achieved by ZCOMP are given in Figure 15. We also show the upper bound cache compression ratio (LimitCC) assuming we can compress cache lines to arbitrary sizes (at a byte granularity), and compress as many lines as possible in a cache set regardless of physical cache line boundaries. This is a policy that could be approached with advanced cache compression architectures, e.g., Skewed Compressed Caches [47]. We also present the average compression ratio for a more practical Two Tag architecture (TwoTagCC) where we can combine at most two logical lines into one physical line [26]. Both cache compression options use the Frequent Pattern Compression with Limited Dictionary (FPC-D) algorithm [9], which extents Frequent Pattern Compression [11] to include a limited dictionary, achieving higher compression ratios at lower latency and complexity.

The geometric mean compression ratio for ZCOMP is 1.8, while it is only 1.1 for the TwoTagCC architecture, and 1.54 for unrestricted cache compression architecture (LimitCC). FPC-D captures more patterns than just zeros. However, we attribute the lower compression ratio of LimitCC to the overhead of the compression prefix which is 8 bytes per cache line in FPC-D [9] compared to only two bytes per cacheline for ZCOMP. The TwoTagCC architecture needs to compress two cache lines in the space of one physical line, which requires lines in the same set to have complementary compressed lengths. This is much harder to achieve when the average compressed line size is lower than 2X.

6 RELATED WORK

Multiple prior works have targeted DNN sparsity. Most of these efforts focus on model pruning and quantization [29, 30, 41, 54, 55]. Model based optimizations reduce memory usage of weights by removing them or using reduced precision. However, we observe that weights correspond to only a small portion of the memory usage.

There are prior works on zero-skipping and activation map compression to reduce memory transfers [12, 19, 44]. These efforts primarily target hardware accelerators with fully customized datapaths and memory hierarchies which avoid the challenges of interacting with automated cache hierarchies and virtual memory systems. For GPUs, CDMA leverages feature map sparsity to reduce CPU-GPU data transfers using a customized DMA [46]. DeftNN performs data fission between multiple elements to reduce data movements [33].

Large memory footprints originating from feature maps have been identified by prior work. Virtualized DNN uses CPU memory to virtually expand the memory capacity of GPUs to offload feature maps [38, 45]. Fused-layer CNN accelerator merges multiple layers to minimize intermediate data buffering [13]. Another prior work proposes to discard some feature maps and re-compute them during backward pass to reduce overall memory footprints [16].

Several other DNN accelerator systems which are FPGA-based [24, 40, 50], ASIC-based [8, 18, 35, 36], or using processing-in-memory [21, 25, 49] have been proposed. CPU-based deep learning optimizations mainly center around improving the SIMD compute capability, data layouts and parallelization [27, 39, 52, 53].

7 CONCLUSIONS

Most memory usage in DNN training originates from the cross-layer feature map transfers, which are often sparse. Although hardware accelerators leverage this sparsity to compress feature maps using fully-customized datapaths and memory hierarchies, it is challenging for general-purpose processors to efficiently capture this opportunity due to automated caches, coherence fabric and virtual memory boundaries.

In this paper, we introduced ZCOMP vector ISA extensions to mitigate cross-layer memory footprint for CPUs. Targeted for bulk sequential feature map compression, ZCOMP instructions provide high-throughput parallel execution, transparent interaction with cache hierarchy and virtual memory, and a flexible software interface. On full networks, compared to an uncompressed baseline, ZCOMP reduces average memory traffic by 31% and 23% for training and inference, respectively, leading to an average 11% and 3% performance improvements. This performance improvement is much higher than the 4% (training) and -2% (inference) achieved by AVX512 vector extensions.

REFERENCES

[1] [n. d.]. CLOUD TPU. https://cloud.google.com/tpu/.

[2] [n. d.]. DeepBench, Baidu. https://github.com/baidu-research/DeepBench.

[3] [n. d.]. Intel Intrinsics Guide. https://software.intel.com/sites/landingpage/IntrinsicsGuide/.

[4] [n. d.]. Intel Math Kernel Library (MKL). https://software.intel.com/en-us/mkl.

[5] 2018. Intel Architecture Instruction Set Extensions and Future Features Programming Reference. https://software.intel.com/sites/default/files/managed/c5/15/architecture-instruction-set-extensions-programming-reference.pdf.

[6] 2018. NVIDIA TURING GPU ARCHITECTURE. https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf.

[7] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. [n. d.]. Tensorflow: a system for large-scale machine learning.

[8] Vahideh Aklaghi, Amir Yazdanbakhsh, Kambiz Samadi, Rajesh K. Gupta, and Hadi Esmaeilzadeh. 2018. Snapea: Predictive early activation for reducing computation in deep convolutional neural networks. In *ACM/IEEE Int. Symposium on Computer Architecture*.

[9] Alaa R Alameldeen and Rajat Agarwal. 2018. Opportunistic Compression for Direct-Mapped DRAM Caches. In *Proceedings of the 4th Annual International Symposium on Memory Systems*. 129–136.

[10] Alaa R Alameldeen and David A Wood. 2004. Adaptive Cache Compression for High-Performance Processors. In *Proceedings of the 31st ACM/IEEE Annual International Symposium on Computer Architecture, Munich, Germany*. 212–223.

[11] Alaa R Alameldeen and David A Wood. 2004. Frequent Pattern Compression" A Significance-Based Compression Scheme for L2 Caches. *University of Wisconsin Department of Computer Sciences Technical Report 1500* (April 2004).

[12] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. 2016. Cnvlutin: Ineffectual-neuron-free deep neural network computing. In *ACM SIGARCH Computer Architecture News*, Vol. 44. IEEE Press, 1–13.

[13] Manoj Alwani, Han Chen, Michael Ferdman, and Peter Milder. 2016. Fused-layer CNN accelerators. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, 22.

[14] A Arelakis and Per Stenstrom. 2014. SC2: A Statistical Compression Cache Scheme. In *Proceedings of the 41st ACM/IEEE Annual International Symposium on Computer Architecture, Minneapolis, MN*. 145–156.

[15] Trevor E. Carlson, Wim Heirman, Stijn Eyerman, Ibrahim Hur, and Lieven Eeckhout. 2014. An Evaluation of High-Level Mechanistic Core Models. *ACM Transactions on Architecture and Code Optimization (TACO)*, Article 5 (2014), 23 pages. https://doi.org/10.1145/2629677

[16] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174* (2016).

[17] X Chen, Yang L, R.P Dick, L Shang, and H Lekstsas. 2010. C-Pack: A High-Performance Microprocessor Cache Compression Algorithm. *IEEE Transactions on VLSI Systems* 18, 8 (2010), 1196–1208.

[18] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, et al. 2014. Dadiannao: A machine-learning supercomputer. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 609–622.

[19] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. 2016. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *ACM SIGARCH Computer Architecture News*, Vol. 44. IEEE Press, 367–379.

[20] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759* (2014).

[21] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. 2016. Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory. In *ACM SIGARCH Computer Architecture News*, Vol. 44. IEEE Press, 27–39.

[22] Julien Dusser, Thomas Piquet, and Andre Seznec. 2009. Zero-Content Augmented Caches. In *Proceedings of 23rd International Conference on Supercomputing*. 46–55.

[23] Agner Fog. 2018. Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs. (September 2018).

[24] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, et al. 2018. A configurable cloud-scale DNN processor for real-time AI. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*. IEEE Press, 1–14.

[25] Mingyu Gao, Jing Pu, Xuan Yang, Mark Horowitz, and Christos Kozyrakis. 2017. Tetris: Scalable and efficient neural network acceleration with 3d memory. *ACM SIGOPS Operating Systems Review* 51, 2 (2017), 751–764.

[26] Jayesh Gaur, Alaa R Alameldeen, and Sreenivas Subramoney. 2016. Base-Victim Compression: An Opportunistic Cache Compression Architecture. In *Proceedings of the 43rd ACM/IEEE Annual International Symposium on Computer Architecture, Seoul, South Korea*. 317–328. https://doi.org/10.1109/ISCA.2016.36

[27] Evangelos Georganas, Sasikanth Avancha, Kunal Banerjee, Dhiraj Kalamkar, Greg Henry, Hans Pabst, and Alexander Heinecke. 2018. Anatomy of high-performance deep learning convolutions on SIMD architectures. *arXiv preprint arXiv:1808.05567* (2018).

[28] Erik G Hallnor and Steven K Reinhardt. 2005. A Unified Compressed Memory Hierarchy. In *Proceedings of the 32nd ACM/IEEE Annual International Symposium on Computer Archtitecture, Madison, WI*. 201–212.

[29] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. 2016. EIE: efficient inference engine on compressed deep neural network. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*. IEEE, 243–254.

[30] Song Han, Huizi Mao, and William J Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149* (2015).

[31] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, et al. 2018. Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective. In *High Performance Computer Architecture (HPCA), 2018 IEEE International Symposium on*. IEEE, 620–629.

[32] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[33] Parker Hill, Animesh Jain, Mason Hill, Babak Zamirai, Chang-Hong Hsu, Michael A Laurenzano, Scott Mahlke, Lingjia Tang, and Jason Mars. 2017. Deftnn: Addressing bottlenecks for dnn execution on gpus via synapse vector elimination and near-compute data fission. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 786–799.

[34] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*. ACM, 675–678.

[35] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on*. IEEE, 1–12.

[36] Patrick Judd, Jorge Albericio, Tayler Hetherington, Tor M Aamodt, and Andreas Moshovos. 2016. Stripes: Bit-serial deep neural network computing. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 1–12.

[37] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 1097–1105.

[38] Y. Kwon and M. Rhu. 2018. A Case for Memory-Centric HPC System Architecture for Training Deep Neural Networks. *IEEE Computer Architecture Letters* 17, 2 (July 2018), 134–138.

[39] Yizhi Liu, Yao Wang, Ruofei Yu, Mu Li, Vin Sharma, and Yida Wang. 2018. Optimizing CNN Model Inference on CPUs. *arXiv preprint arXiv:1809.02697* (2018).

[40] Eriko Nurvitadhi, Ganesh Venkatesh, Jaewoong Sim, Debbie Marr, Randy Huang, Jason Ong Gee Hock, Yeong Tat Liew, Krishnan Srivatsan, Duncan Moss, Suchit Subhaschandra, et al. 2017. Can fpgas beat gpus in accelerating next-generation deep neural networks?. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 5–14.

[41] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W Keckler, and William J Dally. 2017. Scnn: An accelerator for compressed-sparse convolutional neural networks. In *ACM SIGARCH Computer Architecture News*, Vol. 45. ACM, 27–40.

[42] Jongsoo Park, Maxim Naumov, Protonu Basu, Summer Deng, Aravind Kalaiah, Daya Khudia, James Law, Parth Malani, Andrey Malevich, Satish Nadathur, et al. 2018. Deep Learning Inference in Facebook Data Centers: Characterization, Performance Optimizations and Hardware Implications. *arXiv preprint arXiv:1811.09886* (2018).

[43] Gennady Pekhimenko, Vivek Seshadri, Onur Mutlu, Phillip B Kozuch, Michael A adn Gibbons, and Todd C Mowry. 2012. Base-Delta-Immediate Compression: Practical Data Compression for On-Chip Caches. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, Minneapolis, MN*. 51–63.

[44] Brandon Reagen, Paul Whatmough, Robert Adolf, Saketh Rama, Hyunkwang Lee, Sae Kyu Lee, José Miguel Hernández-Lobato, Gu-Yeon Wei, and David Brooks. 2016. Minerva: Enabling low-power, highly-accurate deep neural network accelerators. In *ACM SIGARCH Computer Architecture News*, Vol. 44. IEEE Press, 267–278.

[45] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W Keckler. 2016. vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *The 49th Annual IEEE/ACM International*

Symposium on Microarchitecture. IEEE Press, 18.

[46] Minsoo Rhu, Mike O'Connor, Niladrish Chatterjee, Jeff Pool, Youngeun Kwon, and Stephen W Keckler. 2018. Compressing DMA Engine: Leveraging Activation Sparsity for Training Deep Neural Networks. In *High Performance Computer Architecture (HPCA), 2018 IEEE International Symposium on*. IEEE, 78–91.

[47] Somayeh Sardashti, Andre Seznec, and David A Wood. 2014. Skewed Compressed Caches. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. 331–342.

[48] Somayeh Sardashti and David A Wood. 2013. Decoupled Compressed Cache: Exploiting Spatial Locality for Energy-Optimized Compressed Caching. In *Proceedings of the 46th Annual International Symposium on Microarchitecture, Davis, CA*.

[49] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R Stanley Williams, and Vivek Srikumar. 2016. ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. *ACM SIGARCH Computer Architecture News* 44, 3 (2016), 14–26.

[50] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Mishra, and Hadi Esmaeilzadeh. 2016. From high-level deep neural models to FPGAs. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, 17.

[51] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[52] Linpeng Tang, Yida Wang, Theodore L Willke, and Kai Li. 2018. Scheduling Computation Graphs of Deep Learning Models on Manycore CPUs. *arXiv preprint arXiv:1807.09667* (2018).

[53] Vincent Vanhoucke, Andrew Senior, and Mark Z. Mao. 2011. Improving the speed of neural networks on CPUs. In *Deep Learning and Unsupervised Feature Learning Workshop, NIPS 2011*.

[54] Jiecao Yu, Andrew Lukefahr, David Palframan, Ganesh Dasika, Reetuparna Das, and Scott Mahlke. 2017. Scalpel: Customizing dnn pruning to the underlying hardware parallelism. In *ACM SIGARCH Computer Architecture News*, Vol. 45. ACM, 548–560.

[55] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. 2016. Cambricon-x: An accelerator for sparse neural networks. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, 20.