Accelerating Deep Learning via Architectural Support for Efficient Sparse Data Computation and Storage

ABSTRACT

Deep learning has recently emerged as an important machine learning approach, with big deep neural networks (DNN) models trained on vast amounts of data demonstrating state-of-the-art accuracy on important yet challenging artificial intelligence tasks, such as image and speech recognition. However, training big DNN models using large training data is both compute and memory intensive.

This paper proposes hardware techniques for improving system performance and scalability for DNN training workloads by exploiting the sparse nature of computation and data to reduce the compute and memory requirements. Our techniques improve training efficiency by avoiding resource utilization on sparse data values (i.e., zeroes) which do not impact training quality. Our design is transparent to software, enabling existing codes to enjoy a performance boost without software modifications.

Evaluation on real and simulated hardware using real-world image recognition workloads shows that our low-cost techniques can significantly improve DNN training performance and outperform state-of-the-art software approaches.

1. INTRODUCTION

Deep learning has recently attracted significant attention because of the state-of-the-art performance of deep neural networks (DNNs) on important but challenging artificial intelligence tasks, such as image recognition [1, 2, 3, 4], speech recognition [5, 6, 7], and text processing [8, 9, 10]. A key driver of these machine learning advancements is the ability to train big DNN models (billions of parameters) using large amounts of examples (TBs of data). However, the compute and memory resources required to train big models to reasonable accuracy in a practical amount of time (days instead of months) are significantly high, and surpass the capabilities of a single commodity server. Thus, big DNN models are, in practice, trained in a distributed fashion using a cluster of 100s/1000s of servers, leveraging parallel hardware resources [3, 4]. Addressing the high computational costs of DNN training is critical to sustain task accuracy improvements through model and data scaling.

This paper presents a hardware approach that exploits the computation pattern of DNN training to improve performance and scalability by reducing the compute and cache resource requirements. Our approach is based on the observation that training workloads frequently perform multiply-accumulate computations on sparse data (i.e., zeroes), and that such computations can be avoided without affecting the result or harming model quality. We therefore improve training performance by avoiding the compute and memory system consumption of sparse data and associated computations. The benefits of our approach are larger for big DNN models which typically oversubscribe system resources (e.g., bandwidth).

Figure 1 illustrates a high-level comparison of processor and memory system utilization in a conventional (left) and our proposed (right) system. Resource utilization on zeroes (word granularity in the processor and cache line granularity in the memory system) are white, while those used on other values are shaded grey. Compared to a conventional system, our proposed system eliminates or greatly reduces the resource consumption for computations on and storage of zero data to improve the performance of computations on and and storage of useful data.

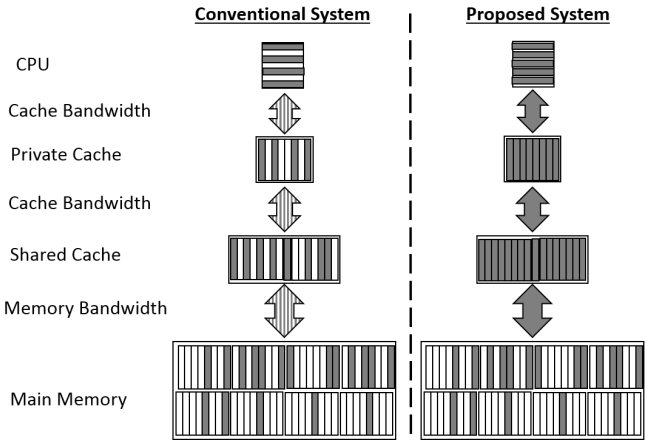

Figure 1: Processor and memory system utilization of sparse (white) and non-sparse (shaded) data.

Our proposed optimizations can improve DNN training peformance and scalability in three ways. First, by eliminating computations on useless data while processing a training example, more iterations over the data set can be completed within a time budget (e.g., a week), which can improve model quality. Second, memory system bandwidth is a key bottleneck for *data-parallelism* within a machine (i.e., processing multiple examples at once using multiple CPU cores). By reducing the bandwidth utilization for each example, data-parallelism can scale training throughput more

effectively. Finally, a standard approach for fitting big DNN models into the last level cache is partitioning the model across multiple machines to exploit the aggregate cache capacity (a.k.a., *model-parallelism*). By reducing the cache consumption of a model partition, our techniques can increase the per-machine partition size and reduce the number of machines required to achieve a target training throughput, reducing model parallelism costs.

Prior software [11, 12] and hardware [13, 14, 15] approaches for spare data computations are less effective for DNN training for two reasons. First, those techniques assume that sparsity exists only in matrices but not in vectors, whereas in DNN training both matrices and vectors can be sparse. Second, those techniques assume that the sparse data structures are static, so that the cost of constructing a sparse representation is amortized over many uses. However, the matrices and vectors in DNN training change for each training example, thus representation cost is incurred repeatedly, hurting performance.

Our approach consists of processor and memory system techniques for reducing sparse data computation overheads. Our processor extensions are based on "zero-optimizable" instructions, which are instructions whose results are either no longer necessary or can be generated more efficiently because an input data of the program is zero. Our optimizations reduce execution cycles and processor resource pressure by skipping zero-optimizable instructions (including entire loops) or executing them more efficiently. Our memory system extensions efficiently track zero data at cache line granularity to avoid the storage and bandwidth costs of zero cache lines. Our approach is transparent to software and therefore benefits existing binaries.

We evaluate our proposed hardware extensions on real and simulated hardware using real-world DNN training workloads for image recognition. The results show that our optiimizations improve training performance by up to 3*X* for single-threaded training and up to 6*X* for multi-threaded training.

This paper makes the following contributions.

- We propose new hardware techniques for improving the performance and scalability of DNN training by reducing computational requirements of sparse data computations, without requiring software changes. We present a detailed design of our proposed hardware extensions, which add negligible logic on the critical path of processor execution and memory accesses.
- We study the impact of sparse data on computation in a real-world DNN training for an image recognition task. We show that eliminating such unnecessary computations can greatly improve performance and scalability of such a real-world workload.
- We quantitatively evaluate how our optimizations improve DNN training peformance and scalability using standard image recognition workloads. We show up to 6*X* training speedup with low hardware cost.

The rest of the paper is organized as follows. Section 2 provides background on DNN and DNN training. Section 3 studies spare data computations in DNN training. Our processor optimizations are described in Section 4, while our cache optimizations are described in Section 5. We present our evalutation results in Section 6, review related work in Section 7, and conclude in Section 8.

2. BACKGROUND

2.1 Deep Neural Networks

DNNs consist of large numbers of neurons with multiple inputs and a single output called an activation. Neurons are connected hierarchically, layer by layer, with the activations of neurons in layer $l-1$ serving as inputs to neurons in layer l. This deep hierarchical structure enables DNNs to learn complex AI tasks, such as image recognition, speech recognition and text processing [16]. DNNs comprise *convolutional* layers (possibly interleaved with *pooling* layers) at the bottom of the hierarchy followed by *fully connected* layers. Convolutional layers, which are inspired by the visual cortex [17], extract features from input samples, and consist of neurons that are only connected to spatially local neurons in the lower layer [18]. Pooling layers summarize the features learned by convolutional layers (e.g., identify the maximum intensity in a cluster of image pixels, reduce spectral variance in speech samples). Fully connected layers classify the learned features into a number of categories (e.g., handwritten digits) and consist of neurons that are connected to all neurons in the lower layer.

2.2 DNN Training

A common approach for training DNNs is using learning algorithms, such as stochastic gradient descent (SGD) [19], and labeled training data to tune the neural network parameters for a specific task. The parameters are the *bias* of each neuron and the *weight* of each neural connection. Each training input is processed in three steps: *feed-forward evaluation*, *back-propagation*, and *weight updates*.

Feed-forward evaluation: Define a_i as the activation of neuron i in layer l. It is computed as a function of its J inputs from neurons in the preceding layer $l-1$:

$$a_i = f\left(\left(\sum_{j=1}^{J} w_{ij} \times a_j\right) + b_i\right) , \tag{1}$$

where w_{ij} is the weight associated with the connection between neurons i at layer l and neuron j at layer $l-1$, and b_i is a bias term associated with neuron i. The activation function, f, associated with all neurons in the network is a pre-defined non-linear function, typically sigmoid or hyperbolic tangent.

Back-propagation: Error gradients δ are computed for each neuron i in the output layer L:

$$\delta_i = (true_i - a_i) \times f'(a_i) , \tag{2}$$

where $true(x)$ is the true value of the output and $f'(x)$ is the derivative of $f(x)$. These error gradients are back-propagated to each neuron i in the layer l from its S connected neurons in layer $l+1$:

$$\delta_i = \left(\sum_{s=1}^{S} \delta_s \times w_{si}\right) \times f'(a_i) . \tag{3}$$

Weight updates: These error gradients are used to compute the weight deltas, Δw_{ij}, for updating the weights:

$$\Delta w_{ij} = \alpha \times \delta_i \times a_j \quad for\ j = 1...J\ , \tag{4}$$

where α is the learning rate and J is the number of neurons of the layer.

This process is repeated for each input until the entire training data has been processed, which constitutes a training *epoch*. Typically, training continues for multiple epochs, reprocessing the training data set each time, until the error converges to a desired (low) value.

3. SPARSITY IN DNN TRAINING

We motivate our work and project the benefits of our optimizations for DNN training. First, we provide intutition into why sparsity exists in DNN training. Next, using a real-world image recogntion workload, we empirically demonstrate the amount of sparsity that exists in practice. Finally, we discuss how available sparsity can be exploited using our techniques to improve training efficiency.

3.1 Sources of Sparsity

Machine learning experts have long observed that DNN training using back-propagation and gradient descent involves a considerable amount of computation on sparse data structures [20, 21, 1, 22, 23] (i.e., data structures that contain a significant fraction of zeroes). Specifically, performance-critical data of training, such as error gradients and weight deltas, can exhibit noticeable levels of sparsity during training. Some of the sparsity arise naturally from the training algorithm and the underlying matrix multiplication kernels. For example, correct predictions of a neuron's output activation during feed-forward evaluation results in zero-valued neuron error gradients, during back-propagation, which can introduce sparsity in the rest of the network. Beyond this, standard techniques for boosting training quality often introduce additional sparsity in the network. These include techniques such as Rectified Linear Units (ReLUs) [21, 1] for faster convergence, and L_1 [20, 22] and Dropout [23] regularization methods for reducing overfitting.

3.2 Sparsity in Real-word Image Recognition

To provide insight into the amount of sparsity that exists in real-world DNN training workloads, we profile the training of a DNN model on the standard *CIFAR-10* image recognition task [24] (described in 6.1). In our study, we reason about sparsity from two perspectives: (i) computation sparsity and (ii) data sparsity. Computation sparsity measures the percentage of mutiply-add operations that are performed on zero values in the performance-critical phases of training (e.g., feed-forward evaluation), while data sparsity measures the percentage of zeroes in the input data (e.g., activations) of these phases. Both perspectives are useful because they capture different impacts of sparsity on system performance, and motivate different optimization opportunities. Computation sparsity captures the impact on processing cycles, data sparsity captures the impact on memory capacity, and both metrics capture the impact on memory bandwidth. We measure both sparsity metrics over 10 training epochs using the standard training data set of 60000 images.

3.2.1 Computation Sparsity

Figure 2 reports computation sparsity in the key phases of DNN training for the first 10 epochs of training a CIFAR-10 image recognition model. Since computation sparsity represents opportunities to safely reduce processing cycles, Figure 2 reports sparsity for different computation granularities to show the potential benefits for non-vectorized (i.e., *Word*) and vectorized (e.g., *4-Words*) implementations of the computation kernels. For example, for feed-forward evaluation, *Word* sparsity is the percentage of CPU multiply-adds that can be skipped because one of the input activation or weight values is zero, while *N-Words* sparsity is the percentage of N-wide vector (SIMD) multiply-adds that can be skipped because either the N activation values or N weight values are zero.

We make the following four observations regarding computation sparsity in DNN training from Figure 2. First, considerable sparsity exists in all the training phases: *Word* sparsity is 29%—43% for feed-forward evalution, 73%—84% for backpropagation, and 77%–92% for weight updates. Second, the training phases have different amounts of computation sparsity, with feed-forward evaluation having the least amount of computation sparsity. Third, computation sparsity generally *increases* with epoch count. Fourth, vectorization affects computation sparsity of the training phases in different ways: vectorization has no impact on sparsity for backpropagation, but reduces sparsity modestly for weight updates and significantly for feed-forward evaluation (e.g., *4-Words* sparsity is half of *Word* sparsity). In summary, the results shows there is potential to greatly reduce the processing and memory bandwidth requirements of DNN training by exploiting computation sparsity, but vectorization limits the benefits for feed-forward evaluation.

3.2.2 Data Sparsity

Figure 3 reports the sparsity of the different performance-critical data in DNN training: (i) activations, (ii) error gradients, (iii) weight deltas, and (iv) weights (defined in Section 2.2). The results are presented for word and cacheline granularities. Data sparsity at word granularity is the percentage of individual data values (e.g., activations) which are zeroes, while cacheline granularity is the percentage of data cache lines containing only zeroes. Since training data values are represented with 4-byte floats, the 64-byte cacheline granularity represents clusterings of 16 sparse data values. Viewing data sparsity at both word and cacheline granularities helps one understand the trade-offs of different optimization strategies since significantly more expensive hardware is required to track sparsity at word granularity compared to cacheline granularity.

We make the following three observations regarding data sparsity in DNN training from Figure 3. First, sparsity level varies across the different data items: at word granularity, weights have 0% sparsity (i.e., dense), while activations (26%—33%), error gradients (83%—85%), and weight deltas (66%—83%) are considerably sparse. Second, for the sparse data items, sparsity tends to increase with more training epochs. Third, sparsity levels are reduced at cacheline granularity to about half of word granularity, which suggests poor clustering of sparse data values in the data layout of the work-

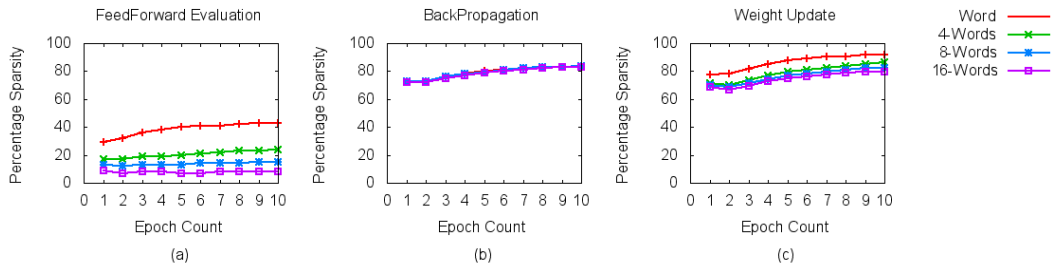

Figure 2: Computation sparsity in CIFAR-10 image recognition training.

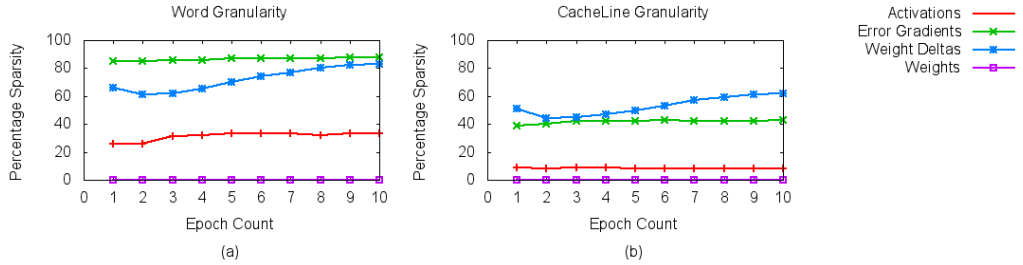

Figure 3: Data sparsity in CIFAR-10 image recognition training.

load. However, considerable levels of sparsity *still* remain at the cacheline granularity. In summary, the results show that cache capacity and bandwidth requirements of DNN training can be greatly reduced by exploiting data sparsity, even at the relatively large cacheline granularity.

3.3 Optimization Opportunities

We motivate our hardware optimizations for sparse computations and data in DNN training by studying a performance-critical kernel for computing weight deltas in back-propagation. Figure 4 illustrates a simplified version of this kernel.[1] The weight deltas of a layer in the DNN are computed by the inner product of the neuron activations and error gradients. Given the amounts of computation and data sparsity in training, a promising optimization for this kernel is to skip the multiply-add operations if either *activations[i]* or *errors[j]* is zero. Moreover, if *errors[j]* is zero, the inner loop can be skipped entirely since *errors[j]* is loop-invariant. These optimization ideas also apply to the other performance-critical kernels of training, e.g., feed-forward evalution.

```
// deltas[] is appropriately initialized
for (int j = 0; j < OUTPUT_COUNT; j++){
 for (int i = 0; i < INPUT_COUNT; i++) {
   int k = j * INPUT_COUNT + i;
   deltas[k] += activations[i] * errors[j];
 }
}
```

Figure 4: Code snippet for computing weight deltas.

Although these optimizations could be implemented in software by checking the data values for zero and guarding computations based on those checks, such a software approach has a couple of practical limitations. First, it requires software changes which might not be possible for existing binaries. Second, the required software checks incur both computation and memory overheads, which could be significant and outweigh the optimization benefits. For example, checking *activations[i]* and checking *errors[j]* for zeroes have different performance impact. Checking *errors[j]* for zeroes is likely to be beneficial because it can be done *outside* the inner loop, and helps to skip large amounts of computation. In contrast, checking *activations[i]* for zeroes will likely hurt performance because it occurs *inside* the inner loop, and can save only a small amount of computation.

3.4 Overview of our Hardware Optimizations

In this work, we propose hardware optimizations for improving DNN training performance by exploiting sparsity in performance-critical computations and data of training workloads. Our harware mechanisms are designed to avoid the limitations and overheads of a software approach. Our approach consists of low cost extensions to the processor and memory systems. Our processor extensions exploit computation sparsity to *safely* skip instructions that compute on sparse data thereby reducing the compute requirements of DNN training (Section 4). Our memory system extensions exploit data sparsity to efficiently store and transfer sparse data through a parallel cache hierachy (a.k.a., *Zero* caches) thereby reducing the memory requirements of DNN training (Section 5). In our evaluation, we compare the performance benefits of software and hardware approaches for exploiting computation and data sparsity in real-world DNN training workloads (Section 6).

[1] Our approach also applies to vectorized (e.g., SIMD) kernels, but we use the simple forms in our discussion for convenience.

4. PROCESSOR OPTIMIZATIONS

Our processor extensions for exploiting computation and data sparsity in DNN training comprise of frontend and backend extensions in the modern processor pipeline. The key idea is to optimize existing training loop codes based on speculating that input data will be sparse and executing the optimized loop codes when that happens. Given the high levels of sparsity in a nunmber of performance critical portions of DNN training (Figures 2 and 3), we expect that most of the execution time will be spent in the optimized loop codes, thus improving training performance greatly. An overview of our processor extensions is presented in Figure 5.

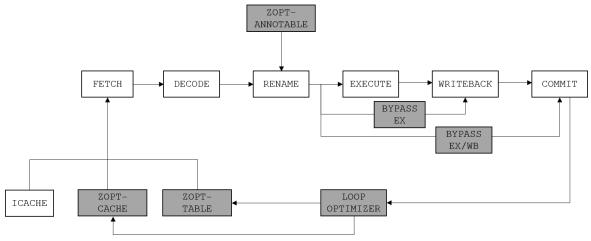

Figure 5: Overview of processor extensions.

4.1 Back-End Extensions

Our backend extensions perform two critical functions in the optimization of DNN training code: (i) creating optimized versions of the loops in the training code, and (ii) enabling the frontend to safely execute the created optimized loop codes for better performance. Our extensions discover loops to optimize by tracing the stream of committed instructions. The optimized loop codes are kept in a special instruction cache, called the *ZOptCache*. We expect the *ZOptCache* to be small because there are only a handful (four) kernel loops in DNN training. We also leverage our cache optimizations described in Section 5 to detect when data values are loaded from a zero cacheline.

4.1.1 Creating Optimized Loop Codes

The goal of the optimizer is to generate more efficient versions of training code loops that can boost training performance when executed in place of the original loop codes. Efficient versions of a loop are created through aggressive loop optimizations based on assumed sparsity in the loop's data and computation. Multiple optimized code versions could be created for a given loop depending on the different sparsity assumptions. Thererfore, the execution of optimized loop codes must be predicated on the underlying assumptions being true. To enable this, the optimizer tags each generated code with description of the conditions that guard its safe execution. Typical examples of such conditions include the requirement that a particular input data is zero, or that it was loaded from a cacheline containing only zero data values.

We use the snippet of training code for computing weight deltas in Figure 4 to illustrate how our optimizer generates efficent versions of training code loops. A simplified machine code sequence corresponding to Figure 4 is presented in Figure 6 as a control flow graph (CFG) of three basic blocks.[2] The inner loop of Figure 4 is represented by the middle block (labelled *BB2*), while the outer loop is represented by blocks *BB1* and *BB3*. The loop-invariant input, *errors[j]*, in Figure 4 is represented by R1, while the loop-variant inputs, *activations[i]* and *deltas[k]*, are represented by R3 and R5 respectively. R2 and R7 correspond to the loop counters of the outer and inner loops respectively.

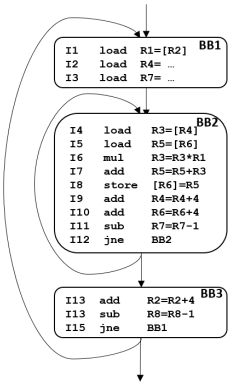

Figure 6: Machine code corresponding to the weight deltas computation source code in Figure 4.

The key idea of our loop optimizations is to identify input data in the loop, such that if the data was zero it would become possible to safely skip some other instructions or execute those instructions more efficiently. We refer to the input data that enables optimizations as the *anchor* input, and the instructions that can be optimized when the anchor input is zero as zero-optimizable instructions. For example, if R1 was zero in Figure 6, instruction I6 in *BB2* could be executed more efficiently (i.e., set to zero), while I4, I5, I7, and I8 could all be skipped. This is because, when R1 is zero, R3 is set to zero by I6 irrespective of the value loaded in I4, meaning the I4 can be skipped. Moreover, I7 can be skipped because does not change R5 since R3 is zero, and thus the following store I8 will write back the same value loaded from memor by I5, meaning that all three instructions can be skipped. Since all the optimized instructions execute in an inner loop, this optimization is likely to greatly improve performance, and so this simple example demonstrates the effectiveness of that exploiting computation and data sparsity in loops. As discussed in Section 3.3, a loop can have multiple anchor inputs, each with different performance benefits, and so, for better coverage we create multiple optimized versions of a loop, for different anchor inputs.

Types of Optimized Loop Codes.

The manner in which an optimized loop code is created depends on static and dynamic properties of the anchor input. The static property is whether or not the anchor input is loop-invariant, and the dynamic property is whether or not the input is clustered with other input values that are zero (e.g., in a zero cacheline). Since loop-invariant anchor inputs enable different (and more) optimization opportunities, we create optimized loops of loop-invariant anchor inputs differ-

[2]Our example in Figure 6 is unoptimized for simplicity of explanation.

ently than for loop-variant anchor inputs. However, for good coverage, we construct two optimized loops for each anchor input to handle both cases of when it is a standalone zero value or clustered with other zero data values. Thus, as discussed below, we create four types of optmized loop codes for: (i) clustered loop-invariant anchor inputs, (ii) standalone loop-invariant achor inputs, (iii) clustered loop-variant anchor inputs, and (iv) standalone loop-variant anchor inputs.

Figure 7 illustrates how we create optimized loop codes for loop-invariant anchor inputs, using R1 in Figure 6 as the example anchor input. Figure 7(a) shows the optimization for a standalone anchor input, in which execution is steered into an optimized code block (*OB1*) after one iteration of *BB2*. This is the optimization example we can studied earlier, in which instructions I4—I8 could either skipped or executed more efficiently in each iteration of the loop. Block *OB1* executes in place of the remaining iterations of *BB2* and ensures that the loop exit invariants are satisfied on entry into *BB3*. Figure 7(b) shows the optimization for when R1 is in a cluster of zero data values, and specifically when R1 is the first word in a cacheline of zero data values. In this case, execution is steered into *OB2* which executes in place of the remaining 15 iterations of *BB2* (corresponding to the other R1 values in the zero cacheline), before returning control to *BB1*. These two examples show that optimizing for loop-invariant zero data can greatly reduce execution cycles of DNN training loops.

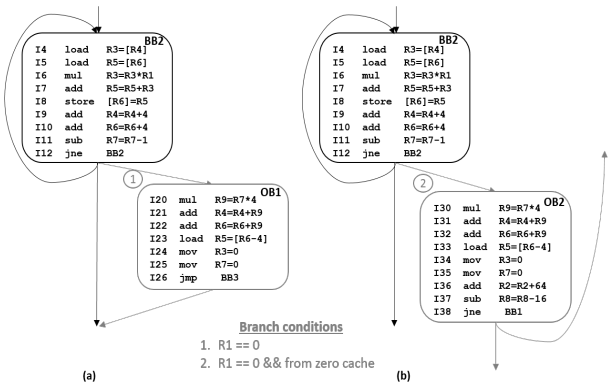

Figure 7: Optimizing for loop-invariant zero data (R1): (a) for data from data cache, redirect execution to skip inner loop, and (b) for data from zero cache (i.e., cluster of 16 **zero data values) redirect execution to skip 16 executions of inner loop.**

Figure 8 illustrates how we create optimized loops for loop-variant anchor inputs, using R3 in Figure 6 as the example anchor input. We don't create new code for standalone loop-variant anchor inputs, rather, as shown in Figure 8 , we generate code annotations that direct the frontend on how to cheaply optimize the code sequence when the anchor is zero. In this case, if R3 is zero, then in the current iteration I5, I7, and I8 can be skipped, and I6 efficiently executed (similar to if R1 is zero). For the frontend, skipping an instruction simply means removing it from the processor pipeline (e.g., by committing without execution) and making its allocated resources available immediately. Code annotations are maintained for each annotated instruction in a special struture called *ZOpt-AnnoTable*, which the frontend can quickly access. In each stage of the pipeline, the frontend checks the *ZOpt-AnnoTable* to see if any special processing step has being requested for the instruction. Figure 8(b) shows the optimization for when R3 is in a cluster of zero data values. In this case, execution is directed into an optimized block (*OB3*) to execute in place of the next "N" iterations of *BB2*, where "N" is minimum of the loop counter R7, and the cluster size of zero data values.

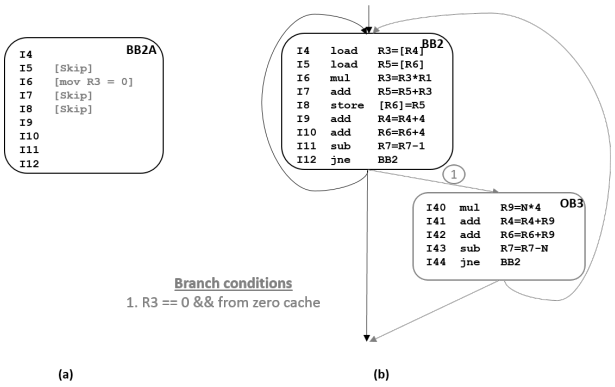

Figure 8: Optimizing for loop-variant zero data (R3): (a) annotate the code block with actions to be taken by frontend, and (b) redirect execution for data from zero cache.

4.1.2 Enabling Execution of Optimized Loop Codes

The primary way that the optimized loop codes get executed is by redirecting the back edge of a loop into the most profitable optimized code that is safe to execute. This means that at least one iteration of the original loop is always executed before optimized loop codes are executed. For loops with hundreds (or thousands) of iterations, such as in DNN training workloads, the performance loss is negligible.

Since instruction fetching takes place in the frontend, the backend maintains a table (a.k.a. *ZOptTable*) that maps each loop address in the training code to the set of optimized versions that have being created for it, as well as the conditions under which each optimized version can be executed. The execution conditions are expressed as invariants on register values as shown in Figures 7 and 8. The *ZOptTable* also indicates whether the backend has generated code annotations in the *ZOpt-AnnoTable* for the frontend to process while executing the optimized code block. Thus, on a backward jump targetting a loop in the training code, the frontend can efficiently and safely steer execution into optimized code for better performance by checking the *ZOptTable* for optimized versions of the loop and accessing the register files to check for the execution prerequisites. We expect the *ZOptTable* to be relatively small (less than 100 entries) that can be checked quickly to avoid unnnecessary delays on jump instructions. After it finds an optimized loop code that it can execute, the frontend fetches the instructions from the *ZOptCache*.

4.2 Front-End Extensions

The frontend extensions of our optimizations are quite

minimal to avoid unnecessary critical path delays. Besides, being responsible for steering executing into optimized loop codes, the frontend also processes code annotations generated by the backend, and tracks data values that loaded from zero cachelines. The actions requested in the code annotations are easily for the frontend to perform, such as removing an instruction from the pipeline or setting a register to a constant value (typically zero). The frontend tracks values that are loaded from zero cachelines through an extra bit the register file. This bit is set for a destination register if a load request is satified by the zero cahce hierarchy and cleared otherwise. Also, the bit value is propagated and updated appropriately in the event of register copies and writes.

5. CACHE OPTIMIZATIONS

Our cache optimizations for DNN training are based on the sparse nature of the performance critical data (in Figure 3). Our approach improves cache performance through a compact representation of cache lines containing only zeroes (a.k.a. *zero cache lines*) in the caches, which helps to avoid the normal bandwidth and storage costs of zero cache lines. These optimizations enable efficient scaling of model size and training threads.

Managing zero data at cache line granularity enables implementation of our optimizations through simple and efficient extensions of existing memory systems. Our design comprises of new mechanisms for: (i) compact representation of zero cache lines, (ii) a decoupled cache hierarchy for zero cache lines, and (iii) tracking zero cache lines in the memory system. We describe these mechanims in the rest of this section.

5.1 Zero Cache Line Representation

Our compact representation exploits the fact that the data bytes of a zero cache line are not required to represent the line in cache. The cache tag is sufficient for this purpose. Also, it is not neccesary to transfer the data bytes of a zero cache line across the caches since they can be synthesized in the processor (read) or main memory (on a writeback) as appropriate. However, in the event of a cache hit, we must quickly determine whether it is a zero cache line that is referenced so that the appropriate data transfer, if need (i.e., non-zero), is done promptly. We consider two alternatives for handling this: (i) an extra bit in the cache tag to identify a zero cache line or (ii) a decoupled hierarchy of cache tags for zero cache lines. Although the first option avoids the extra cost of zero cache line tags, the data store space of zero cache lines goes unused. To avoid this waste, we adopt the second option in our current work.

5.2 Hierarchy of Zero Cache Lines

Figure 9 illustrates a memory system that is augmented with a cache hierarchy for zero cache lines, which we call the *zero cache* hierarchy. The zero cache hierarchy is a multi-level structure with caches (a.k.a., *zero caches*) containing tags but no data bytes. Since zero cache lines are not maintained in the conventional data caches, both cache hierarchies are mutually exclusive. The zero cache hierarchy and the data cache hierarchy have the same number of levels, and can additionally share other properties, such as number of entries, ways, associativity, replacement policies, etc. The coherence of zero caches is maintained across cores using the same protocol as the data caches.

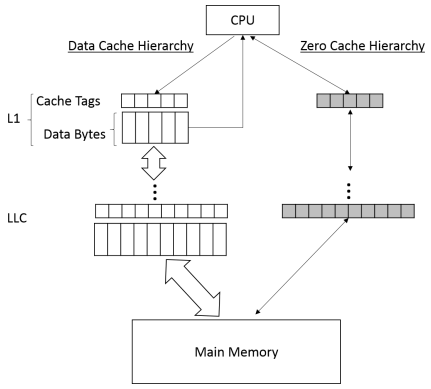

Figure 9: A Memory System with Zero Cache Hierarchy.

Data access requests from the processor are satisfied by accessing the two cache hierarchies in parallel to avoid introducing extra latency. Figure 10 shows the processing of a read request by the *N*th level caches. The request is processed in parallel by the data and zero caches, and forwarded to the next level if it is a miss in both. If the request is a hit in either cache, then the appropriate response is sent to the processor or lower levels of the cache hierarchy. The data cache responds, as normal, with the requested data bytes (or cache line), while the zero cache responds by signaling a zero cache line hit.

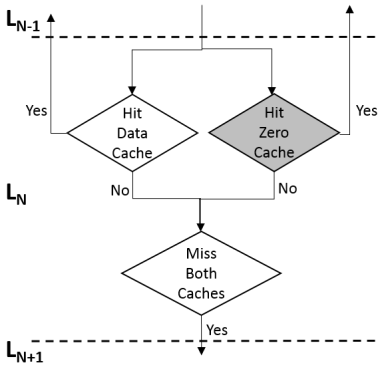

Figure 10: Handling read requests.

5.3 Tracking Zero Cache Lines

Our optimizations are based on the invariant that cache lines reside in the appropriate cache hierarchy: zero cache lines in zero caches and other cache lines in the data caches. To maintain this invariant, we track the zero status of cache lines to ensure that a cache line is placed in the right hierarchy in the following events: (i) update by processor writes, (ii) cache fill from main memory, and (iii) writebacks from lower level caches (e.g., due to evictions). Our tracking operations do not increase cache access latencies as they execute off the critical path of cache accesses. We leverage zero de-

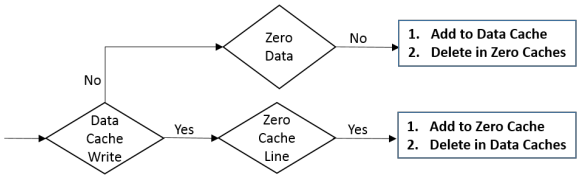

Figure 11: Handling processor writes.

tector hardware [25] to detect that an entire cache line (i.e., 32/64 bytes) is zero.

5.3.1 Processor Writes.

The zero-status of a cache line can be changed by a processor write depending on the current status and write data. The following four situations could arise: (i) write zeroes to a zero cache line, (ii) write non-zeroes to a non-zero cache line, (iii) write non-zeroes to a zero cache line, and (iv) write zeroes to a non-zero cache line. The first two situations are irrelevant since the non-zero status of the cache line is unchanged. Writing a non-zero value to a zero cache line moves the cache line to the data cache in the the same level, and removes the cache line from the zero cache hierarchy. This may require data cache evictions to accomodate the new cache line. Writing zeroes to a non-zero cache line moves the cache line from the data caches into the zero-cache hierarchy if the cache line contains only zeroes after the update. Naturally, the cache tag is moved as well. Figure 11 illustrates how cache updates by processor writes are handled to ensure that cache lines reside in the right hierarchy.

5.3.2 Cache Fills from Main Memory.

Since our optimizations are focused on the cache capacity and bandwidth, the data bytes of zero cache lines are stored in main memory, similar to other cache lines. We extend the memory controller to avoid sending data bytes when handling a cache fill request for a zero cache line. Requests for non-zero cache lines are handled normally.

5.3.3 Writebacks from Lower Level Caches.

Our decoupled cache hierarchies approach implicitly handles writebacks from lower level caches because the zero-status of a cache line is unchanged. Thus, the data caches are not involved by zero cache writebacks, and vice versa.

6. EVALUATION

We now evaluate the effectiveness of our processor and memory system optimizations for DNN training. We conduct our evaluations in terms of computation sparsity and data sparsity.

6.1 Methodology

Real-world Image Recognition Workloads:.

We used image recognition workloads to evaluate the impact of our optimizations on DNN training performance. Image recognition models cover an important class of challenging AI problems, and training them to reasonable task accuracy requires copious amounts of compute and memory cycles [1, 2, 3, 4]. We used four real-world image recognition benchmarks in our experiments: (i) *MNIST* [26], (ii) *CIFAR-10* [24], (iii) *ImageNet-1K* [27], and (iv) *ImageNet-22K* [27]. For our experiments, we replicate the architectures of a high quality (or the best performing) model for each task based on prior work.

- MNIST [4]: The task is the classification of 28x28 grayscale images of handwritten digits into 10 categories (i.e., 0—9). The model for this task is relatively small, containing about 2.5 million in five layers.
- CIFAR-10 [23]: The task is the classification of 32x32 color images into 10 categories. The model for learning this task is moderately large, containing 28.5 million connections in five layers.
- ImageNet-1K [1] & ImageNet-22K [4]: The tasks are the classification of 256x256 color images into 1000 and 22000 categories respectively. The ImageNet-1K model is quite large and contains about 65 million connections in seven layers. The ImageNet-22K model is one of the largest image models available, and contains about 2 billion connections in 8 layers.

Experimental Approach.

We evaluate the performance of our optimizations using a combination of real and simulated hardware experiments. We evaluate our computation sparsity optimizations on real hardware by hand-optimizing the loops in training kernels, for computation sparsity, as would be done by our proposed processor extensions. And in our experiments we alternate execution of a loop and its sparsity-optimized version(s) based on computation and data sparsity information of the model (Section 3.2). Our experimental system is a dual-socket Intel Xeon E2450 CPU, with each socket having eight cores on running at 2.1 Ghz. The memory system comprises of private 32 KB L1 cache, private 256 L2 cache, and a 20MB shared L3 cache on each socket. We evaluate our data sparsity optimizations in a simulation environment by prototyping our proposed zero-cache extensions in the GEM5 simulator [28]. Table 1 presents the simulation parameters.

GEM5 Simulator	
Cache line size	64 bytes
L1 (private)	32KB
L2 (private)	256KB
L3 (shared)	20MB
Coherence protocol	MESI
Memory system	Ruby
RAM size	4GB

Table 1: Memory simulation parameters

DNN Execution Framework.

We measure DNN training performance in our experiments by executing the models in a framework that estimates the performance and scalability of DNN training [29]. The tool is basically a stripped-down version of full-featured training frameworks, such as CAFFE [30] and Adam [4], that

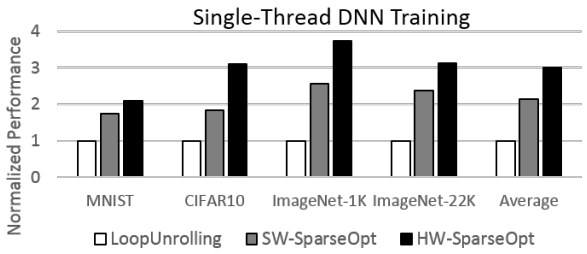

Figure 12: Computation sparsity optimization speedups.

implements only the performance-critical steps of training: feed-forward evaluation, back-propagation, and weight updates. The tool also supports thread parallelism and model parallelism in DNN training. The lighweight nature of the tool makes it convenient to use for performance studies. We adapted the tool for our work by extending it to synthesize sparse input data for each processing step based on the sparsity information that we collected from profiling actual training runs (as described in Section 3.2.2).

Comparison to Software Approaches.

We compare our hardware approach to two software approaches: (i) a baseline approach that (SIMD) vectorizes and unrolls the loops in training kernels [31], and (ii) a software optimization of the baseline that exploits sparsity to dynamically check training data for zeroes in order to skip multiply and addition operations (Section 3.3). We heavily optimize this software approach to mitigate overheads of dynamic checks by applying the optimization only in situations that can yield significant benefits (e.g., skipping inner loop execution). Consequently, we use *SW-SparseOpt* only for back-propagation and weight updates, but not feed-forward evaluation, after empirically verifying that this was the optimal decision. In our results, we refer to the baseline, software-optimized, and hardware-optimized approaches as *LoopUnrolling*, and *SW-SparseOpt*, and *HW-SparseOpt* respectively.

6.2 Computation Sparsity Optimizations

Computation sparsity optimizations improve DNN training performance by dynamically skipping execution on zero data values. We measure the impact of our computation sparsity optimizations on a single-threaded training and multi-threaded training.

6.2.1 Single-Threaded Performance

Figure 12 shows the improvement of single-threaded training performance from computation sparsity optimizations. We make two observations from these results. First, computation sparisty optimizations provide impressive training speedups for all the workloads, and can be effectively exploited by both software and hardware approaches. Second, we that even though *SW-SparseOpt* provides $2X$ average speedup over *LoopUnrolling*, *HW-SparseOpt* achieves the fastest training performance for each workload. On average, *HW-SparseOpt* is $3X$ faster than *LoopUnrolling* and 50% faster than *SW-SparseOpt*.

To understand why *HW-SparseOpt* outperforms *SW-SparseOpt*, we examine the impact of computation sparsity optimizations on the training phases. Figure 13 reports the processing time for a single input in the *LoopUnrolling*, *SW-SparseOpt*, and *HW-SparseOpt* versions of the training phases. The *LoopUnrolling* results show that the three phases contribute roughly equally to overall training time in the baseline case. Recall that *SW-SparseOpt* is used only on back-propagation and weight updates, but not feed-forward evalution, because of the cost/benefit tradeoff. Thus, *SW-SparseOpt* is limited to about $\frac{2}{3}$ of training execution, where it provides significant overhead reduction (up to $10X$ for ImageNet-1K) because of high computation sparsity in back-propagation and weight updates (Figures 2(b) & 2(c)). In comparison, *HW-SparseOpt* achieves similar overhead reductions for back-propagation and weight updates, and can also be applied to feed-forward execution. However, *HW-SparseOpt* achieves much lower overhead reduction in feed-forward evaluation (at most $1.5X$ for ImageNet-1K) because of lower levels of computation sparsity (2(a)). Nevertheless, *HW-SparseOpt* is overall better than *SW-SparseOpt* because feed-forward evaluation becomes the bottleneck after optimizing the other phases, and so that even modest improvements are noticeable in the overall training time.

6.2.2 Multi-threaded Performance

Next we consider the impact of computation sparsity optimizations on multi-threaded training scalability. Figure 14 shows the normalized training performance as we scale the number of training threads up to 16, where the baseline performance is the single-threaded *LoopUnrolling*. We make three observations from these results. First, we see that the benefits of exploiting computation sparsity is sustained and even grows with multi-threading. Second, we see that larger models (ImageNet-1K and ImageNet-22K) which scale poorly due to their large working sets benefit more from computation sparsity optimizations than relatively smaller models (MNIST and CIFAR-10) because of the reduced pressure placed on shared cache capacity and bandwidth by multi-threading. Third, we see that *HW-SparseOpt* continues to outperform *SW-SparseOpt* with multi-threading.

6.3 Data Sparsity Optimizations

We simulate our zero cache design in the GEM5 simulator to evaluate the impact of zero caches on multi-threaded training scalability. The results are presented in Figure 15 in terms of the processing time per input image. *Baseline* is the baseline system without zero caches and corresponding to the simulation parameters in Table 1. *2XCache* is similar to *Baseline* except that the cache sizes are doubled at all the levels. We use *2XCache* to estimate the upper bound performance of doubling the hardware cache cost. *ZeroCache* is our design of augmenting every cache level with a zero cache. As the results show *ZeroCache* did not provide much benefits. Our investigation so far indicates that this might due to the large L3 cache in the system. The fact that *2XCache* is not much better seems to indicate support our current hypothesis. Moreover we observed that *ZeroCache* was indeed improving the hit ratio of *Baseline*, yet this did not translate to much biggger end-to-end wins. We continue

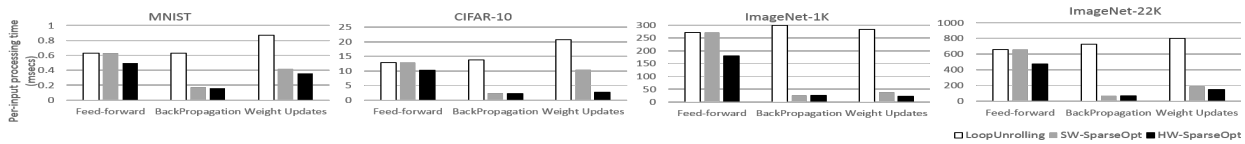

Figure 13: Impact of computation sparsity optimizations on DNN training phases.

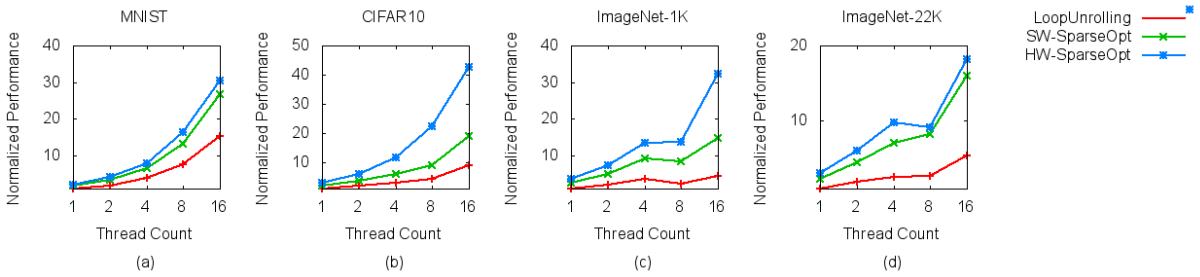

Figure 14: Impact of computation sparsity optimization on DNN training scalability.

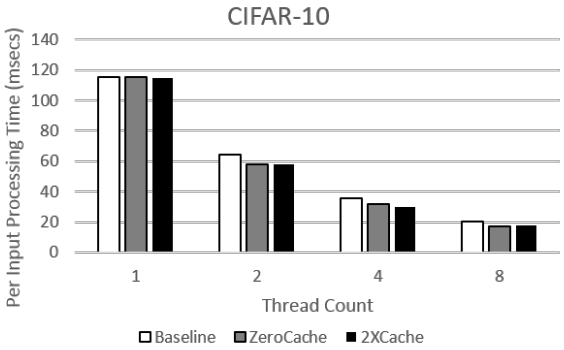

Figure 15: Impact of Data Sparsity optimizations on Multithreaded CIFAR10 Training.

our investigation, but did not have enough time to conclude before the paper deadline.

7. RELATED WORK

To our knowledge this is first work to explore processor and memory system optimizations for exploiting computation and data sparsity to improve DNN training performance in commodity systems. More generally, our work is related to prior work on efficient sparse matrix computations [11, 12, 13, 14, 15, 32], reducing overheads of zero values in the cache [33, 34, 25, 35] and physical register files [36, 37], and dynamic elimination of redundant computations [38, 39, 40, 41].

Prior work on sparse matrix computations, including software [11, 12] and hardware support [13, 14, 15, 32], are unlikely to be efficient for DNN training due to the dynamic nature of the sparse data structures (e.g., error gradients), and relatively lower sparsity levels that what they are turned for ($> 95\%$). Prior work on reducing overheads of zero values in caches [33, 34, 25, 35] treat the zero cache differently from the regular data caches (e.g., not part of coherence protocol, or accepting writes). In contrast, in this work the zero cache is treated exactly like the data caches (save for not having data bytes), and we further consider the impact on multi-threaded workloads. Reducing overheads of zero values in physical register files is complementary to our work. Some of the ideas in our loop optimizations are similar to prior work on dynamically eliminating redundant computations. However, ours is the first to consider these ideas for DNN training workloads.

8. CONCLUSION

In this paper we proposed hardware optimizations for exploiting computation and data sparsity in DNN training to improve training performance. We provide empirical evidence that non-trivial amounts of sparsity exists in real-world DNN training workloads. Our technique includes low-cost extensions to the processor and memory systems for efficiently leverage sparsity optimization opportunities in DNN training. Our evaluation shows that our hardware optimizations enable up to a $3X$ speedup for real-world training workloads compared to a baseline implementation that is vectorized and loop unrolled. Our optimizations are up to $1.5X$ faster a software approach to exploiting sparsity optimization opportunities.

9. REFERENCES

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *NIPS*, 2012.

[2] Q. Le, M. Ranzato, R. Monga, M. Devin, K. Chen, G. Corrado, J. Dean, and A. Ng, "Building high-level features using large scale unsupervised learning," in *ICML*, 2012.

[3] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. W. Senior, P. A. Tucker, K. Yang, and A. Y. Ng, "Large scale distributed deep networks.," in *NIPS*, 2012.

[4] T. Chilimbi, J. Apacible, K. Kalyanaraman, and Y. Suzue, "Project Adam: Building an efficient and scalable deep learning training system," in *OSDI*, 2014.

[5] G. E. Dahl, D. Yu, L. Deng, and A. Acero, "Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition," *Trans. Audio, Speech and Lang. Proc.*, 2012.

[6] G. Hinton, L. Deng, D. Yu, G. Dahl, A. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. Sainath, and B. Kingsbury, "Deep neural networks for acoustic modeling in speech recognition," *Signal Processing Magazine*, 2012.

[7] A. Hannun, C. Case, J. Casper, B. Catanzaro, G. Diamos, E. Elsen, R. Prenger, S. Satheesh, S. Sengupta, A. Coates, *et al.*, "Deepspeech: Scaling up end-to-end speech recognition," *arXiv preprint arXiv:1412.5567*, 2014.

[8] R. Collobert and J. Weston, "A unified architecture for natural language processing: Deep neural networks with multitask learning," in *ICML*, 2008.

[9] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa, "Natural language processing (almost) from scratch," *The Journal of Machine Learning Research*, 2011.

[10] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *NIPS*, 2013.

[11] C. S. Eisenstat, M. Gursky, H. M. Schultz, and H. A. Sherman, "Yale sparse matrix package i: The symmetric codes," *International Journal for Numerical Methods in Engineering*, 1982.

[12] Intel, "Sparse Matrix Storage Formats, Intel Math Kernel Library." `https://software.intel.com/en-us/node/471374`.

[13] J. B. Carter, W. C. Hsieh, L. Stoller, M. R. Swanson, L. Zhang, E. Brunvand, A. Davis, C. Kuo, R. Kuramkote, M. A. Parker, L. Schaelicke, and T. Tateyama, "Impulse: Building a smarter memory controller," in *HPCA*, 1999.

[14] S. Kestur, J. D. Davis, and E. S. Chung, "Towards a universal fpga matrix-vector multiplication architecture," in *FCCM*, 2012.

[15] J. Fowers, K. Ovtcharov, K. Strauss, E. Chung, and G. Stitt, "A high memory bandwidth fpga accelerator for sparse matrix-vector multiplication," in *International Symposium on Field-Programmable Custom Computing Machines*, 2014.

[16] Y. Bengio, "Learning deep architectures for AI," *Foundations and Trends in Machine Learning*, 2009.

[17] D. Hubel and T. Wiesel, "Receptive fields of single neurons in the cat's striate cortex," *Journal of Physiology*, 1959.

[18] Y. LeCun and Y. Bengio, "The handbook of brain theory and neural networks," ch. Convolutional Networks for Images, Speech, and Time Series, 1998.

[19] L. Bottou, "Large-scale machine learning with stochastic gradient descent," in *COMPSTAT*, 2010.

[20] A. Y. Ng, "Feature selection, l1 vs. l2 regularization, and rotational invariance," in *ICML*, 2004.

[21] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines.," in *ICML*, 2010.

[22] Y. Bengio, N. Boulanger-Lewandowski, and R. Pascanu, "Advances in optimizing recurrent networks," in *Proc. ICASSP 38*, 2013.

[23] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, 2014.

[24] A. Krizhevsky, "Learning multiple layers of features from tiny images," Master's thesis, Computer Science Department, University of Toronto, 2009.

[25] J. Dusser, T. Piquet, and A. Seznec, "Zero-content augmented caches," in *ICS*, 2009.

[26] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," in *Proceedings of the IEEE*, pp. 2278–2324, 1998.

[27] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A Large-Scale Hierarchical Image Database," in *CVPR09*, 2009.

[28] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, pp. 1–7, Aug. 2011.

[29] F. Yan, O. Ruwase, Y. He, and T. Chilimbi, "Performance modeling and scalability optimization of distributed deep learning systems," in *KDD*, 2015.

[30] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," *arXiv preprint arXiv:1408.5093*, 2014.

[31] V. Vanhoucke, A. Senior, and M. Z. Mao, "Improving the speed of neural networks on cpus," in *Proc. Deep Learning and Unsupervised Feature Learning NIPS Workshop*, vol. 1, 2011.

[32] V. Seshadri, G. Pekhimenko, O. Ruwase, O. Mutlu, P. B. Gibbons, M. A. Kozuch, T. C. Mowry, and T. Chilimbi, "Page overlays: An enhanced virtual memory framework to enable fine-grained memory management," in *ISCA*, 2015.

[33] L. Villa, M. Zhang, and K. Asanović, "Dynamic zero compression for cache energy reduction," in *MICRO*, 2000.

[34] Y. Zhang, J. Yang, and R. Gupta, "Frequent value locality and value-centric data cache design," in *ASPLOS*, 2000.

[35] M. M. Islam and P. Stenstrom, "Zero-value caches: Cancelling loads that return zero," in *PACT*, 2009.

[36] S. Jourdan, R. Ronen, M. Bekerman, B. Shomar, and A. Yoaz, "A novel renaming scheme to exploit value temporal locality through physical register reuse and unification,"

[37] S. Balakrishnan and G. S. Sohi, "Exploiting value locality in physical register files," in *MICRO*, 2003.

[38] D. A. Connors, H. C. Hunter, B.-C. Cheng, and W.-m. W. Hwu, "Hardware support for dynamic activation of compiler-directed computation reuse," in *ASPLOS*, 2000.

[39] J. D. Collins, D. M. Tullsen, H. Wang, and J. P. Shen, "Dynamic speculative precomputation," in *MICRO*, 2001.

[40] J. A. Butts and G. Sohi, "Dynamic dead-instruction detection and elimination," in *ASPLOS*, 2002.

[41] P. G. Sassone and D. S. Wills, "Dynamic strands: Collapsing speculative dependence chains for reducing pipeline communication," in *MICRO*, 2004.