Efficient Sparse Data Computations for Deep Learning

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, making distributed training on a cluster of server machines, leveraging the aggregate system resources, the standard approach.

This paper proposes hardware techniques for improving system performance and scalability for DNN training workloads by exploiting the sparse nature of computation 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 modifications.

Simulation-based evaluation using standard image recognition workloads shows that our techniques can improve DNN training performance significantly and outperform 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 [29, 31, 14, 8], speech recognition [13, 21, 19], and text processing [10, 11, 34]. 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 [14, 8]. 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 the computation data of training are significantly sparse, and since training kernels are dominated by multiply-accumulate operations, a significant portion of these computations are redundant to the training objective. We therefore improve training performance by avoiding the compute and memory system consumption of sparse data and the associated computations without harming model quality. The performance benefits should be larger for big DNN models where system resources (e.g., bandwidth) are typically oversubscribed.

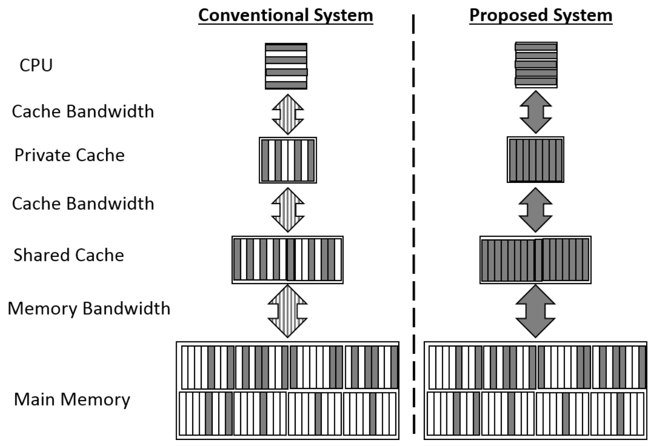


Figure 1: Processor and memory system utilization of sparse (white) and non-sparse (shaded) data.

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 significantly reduces the resource consumption for zero data computations to improve the performance of useful data computations.

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 (and cache) 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 [17, 23] and hardware [7, 27, 18] approaches for sparse matrix-vector multiplications 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 separate techniques for tackling sparse data computation overheads in the processor and memory system. Our processor extensions are based on "zero-optimizable" instructions, which are arithmetic instructions (e.g., multiplication) whose results and side effects are pre-determined when an input operand is a zero. Our optimizations exploit zero-optimizable instructions to reduce execution cycles and processor resource pressure. Our memory system extensions efficiently track zero data at cache granularity in the caches and main memory to avoid the bandwidth costs associated with moving zero cache lines. Our approach does not require software modifications and therefore benefits existing binaries.

We evaluate our proposed hardware extensions in a simulation environment using real-world DNN training workloads for image recognition. The results show that our approach can significantly improve DNN training performance in single threaded, multi-threaded, and model-parallelism scenarios.

This paper makes the following contributions.

- We propose hardware techniques for improving the performance and scalability of DNN training by reducing computational requirements of sparse data computations, without requiring software changes.
- We study the impact of sparse data on computation in a real-world DNN training for an image recognition task.
- 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 quantitatively evaluate how our optimizations improve DNN training peformance and scalability using standard image recognition workloads.

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 [3]. 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 [22], extract features from input samples, and consist of neurons that are only connected to spatially local neurons in the lower layer [32]. 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) [5], 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 terms 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 terms are backpropagated 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 terms are used to compute the weight deltas, Δw_{ij}, for updating the weights:

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

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

In this section we motivate our work and project the benefits of our optimizations for DNN training. First, we provide some intutition of why sparsity exists in DNN training. Next, using an 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 computations on sparse matrices and vectors [36, 35, 29, 4, 41]. The performance-critical data of training are neuron activations and errors (both implemented as vectors) and synaptic weights and corresponding deltas (both implemented as matrices) can be sparse. Some of the sparsity arise naturally from the training process and its matrix-vector multiplication kernel. For example, correct predictions of a neuron's output activation, during feed-forwad evaluation, result in zero-valued neuron error terms, 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) [35, 29] for faster convergence, and L_1 [36, 4] and Dropout [41] regularization methods for reducing overfitting. Trishul to help with this content

3.2. Sparsity in real-word image recognition task

For a better insight into the amount of sparsity in real-world DNN training workloads, we profile training on *CIFAR-10* [28], a standard image recognition task (described in 6.1). In our study, we reason about sparsity from 2 perspectives: (i) data sparsity and (ii) computation sparsity. Specifically, data sparsity measures the amount of zeroes in performance-critical data (e.g., activation vectors), while computation sparsity measures the amount of mutiply-accumulate operations performed on zero values in the main phases of training (e.g., feed-forward evaluation). Both perspectives are useful because they capture different effects of sparsity on system performance. Memory capacity bandwidth impact is captured by data sparsity, processing cycles impact is captured by computation sparsity, and bandwidth impact is captured by both data and computation sparsity. We measure both sparsity metrics over 10 training epochs using a data set of 60000 images.

3.2.1. Data Sparsity. Figure 2(a) illustrates the sparsity of activation and error vectors, and weight delta and weight matrices in CIFAR-10 training. We see that the sparsity amount and rate of change is quite different among the data structures. While the weight matrix is dense, the activation and error vectors and the weight delta matrix are noticeably sparse. We see that sparsity generally increases with training epochs, albeit at varying rates. The error vector has the greatest amount of sparsity (83%—85%), followed by the weight delta matrix (66%—83%), and finally the activation vector (26%—33%). The results show the memory/cache consumption of activations, errors, and weight deltas for this workload can be reduced significantly.

3.2.2. Computation Sparsity. Figure 2(b) reports the computation sparsity of the different training steps. Compared to data sparsity, the results illustrate how the different vectors and matrices are combined through multiplication and addition operations. For example, the sparsity in feed-forward evaluation is the result of multiplying the dense weight matrix and sparse activation vector. We see that considerable sparsity exists in each training step (from 29% for feed-forward evaluation to 92% for computing weight deltas). We also see that the amount of sparsity generally grows with training epochs (e.g., 29%—42% for feed-forward evalution). In summary, these results shows the potential for significant saving in processing cycles by eliminating cycle consumption for generating zero values that do not impact training quality.

3.3. Sparsity at cache line granularity

Our proposed hardware mechanisms track data sparsity at cache line granularity, so it is unlikely that we can fully exploit the amounts of sparsity presented above because the profiling was conducted at a finer granularity (e.g., individual activation values). To get a more accurate view of the effectiveness of our optimizations we repeat the profiling study at a cache line granularity. Since data values are represented as 4-byte floats (or word), each cache line contains up to 16 data values. We measure

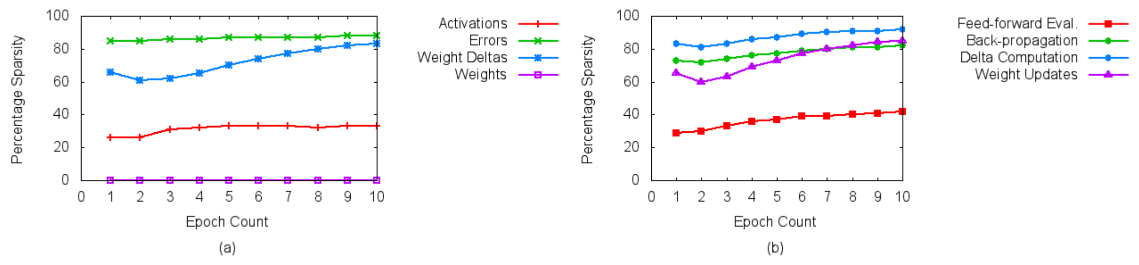


Figure 2: (a) Data and (b) computation sparsity in CIFAR-10 training.

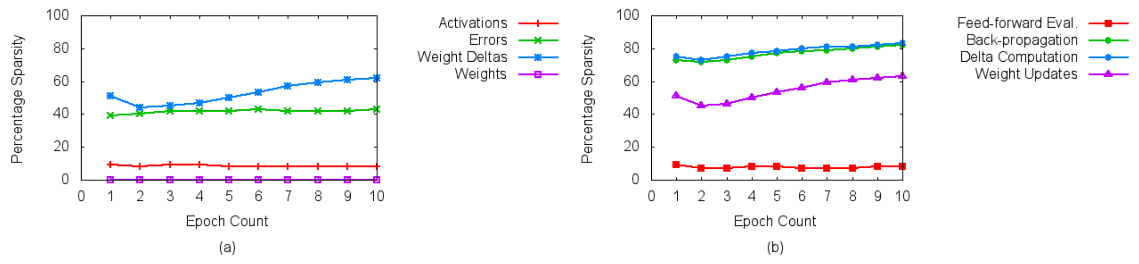


Figure 3: (a) Data and (b) computation sparsity in CIFAR-10 training at cacheline granularity.

sparsity at cache line granularity in the following manner. Data sparsity represents the percentage of cache lines of a data structure that contain only zero values. For computation, we adopt a coarse-grained veiw of computations, i.e., a unit of computation operates on a pair of cache lines (e.g., an activation cache line and a weight cache line in feed-forward evaluation). Thus, computation sparsity represents the percentage of such computations that operate on a sparse cache line. The results are presented in Figures 3(a) and 3(b) for data and computation sparsity respectively.

Figure 3(a) shows that data sparsity at cache line granularity is generally lower compared to word granularity (Figure 2(a), e.g., error sparsity is about half. This indicates that sparse data values are not clustered, which limits the capacity savings achievable by our approach. However, the computation sparsity results in Figure 3(b) presents a more promising picture. We see that computation sparsity at cache line granularity is not much lower than sparsity at word granularity for 3 phases of training: back-propagation, delta computation, and weight updates. This indicates that even though the non-sparse cache line ratio is relatively high, non-sparse cache lines are more likely to be combined with sparse cache lines leading to sparse computations. These results suggest that reducing the cycles and bandwidth consumption of sparse data computations can yield significant performance benefits.

3.4. Optimization Opportunities in Training Codes

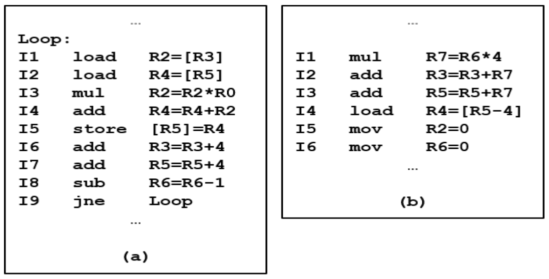
We motivate our hardware optimizations for sparse data and computations in training using a performance-critical kernel for computing synaptic weight gradients during back-propagation. Figure 4 illustrates a simplified version of this kernel[1]. Weight gradients are computed by the inner product of the activation and error vectors. Given the sparsity in training data and computations, we observe that a promising optimization for this kernel is to skip multiply and addition operations if *inputActivations[i]* or *outputErrors[j]* is zero. Moreover, the inner loop can be skipped entirely if *errors[j]* is zero. These optimizations are also effective for other performance-critical kernels of training, e.g., feed-forward evalution.

```
// gradients[] 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;
   gradients[k] += inputActivations[i] * outputErrors[j];
 }
}
```

Figure 4: Code snippet for computing weight gradients.

Although these optimizations could be implemented in software by checking the data values for zero, that 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 compute and memory overheads, which could be significant and outweigh the optimization benefits. For example, checking *inputActivations[i]* and checking *outputErrors[j]* have different performance impacts. Checking *outputErrors[j]* 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 *inputActivations[i]* will likely hurt performance because it occurs inside the inner loop, and can save only a small amount of computation. Our hardware optimizations avoid these

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



```
        ...
Loop:
I1   load    R2=[R3]
I2   load    R4=[R5]
I3   mul     R2=R2*R0
I4   add     R4=R4+R2
I5   store   [R5]=R4
I6   add     R3=R3+4
I7   add     R5=R5+4
I8   sub     R6=R6-1
I9   jne     Loop
        ...

        (a)
```

```
        ...
I1   mul     R7=R6*4
I2   add     R3=R3+R7
I3   add     R5=R5+R7
I4   load    R4=[R5-4]
I5   mov     R2=0
I6   mov     R6=0
        ...

        (b)
```

Figure 5: (a) original and (b) optimized machine code of gradient computation inner loop.

limitations. We compare the performance benefits of software and hardware approaches in our evaluation.

4. Processor Optimizations

Our processor optimizations are based on arithmetic operations, such as addition and multiplication, which are performance critical in training computations, and have predetermined results when one of the input operands is a zero. We refer to machine instructions that perform such arithmetic operations as "zero-optimizable" instructions. Exploiting zero-optimizable instructions to improve training peformance is promising because, as shown in our profiling studies (Figure 2), a significant portion of the inputs to training computations are zeroes.

4.1. Opportunities

We discuss the dynamic optimization opportunities that motivate our processor techniques using the machine code sequences in Figure 5, which correspond to the inner loop of the gradient computation code in Figure 4. Figure 5(a) represents machine code that is generated by a static compiler[2], and consists of five zero-optimizable instructions: I3, I4, I6, I7, and I8. Figure 5(b) shows the code sequence resulting from dynamic optimization of the loop using our techniques when the loop invariant input of I3, R0 (i.e., *outputErrors[j]* in Figure 4), is zero. We now describe how our dynamic techniques improve performance using zero-optimizable instructions with zero input values.

4.1.1. Zero-Optimizable Instructions. A zero-optimizable instruction presents a number of opportunities to increase ILP and reduce resource pressure of training workloads on modern out-of-order processors. These opportunities arise because of the predetermined results of zero-optimizable instructions when computing on zero input operands. Since we focus on zero-optimizable instructions performing additions or multiplications, we consider how these instructions could be affected by a zero input when there are two input and one output operands.

A zero input operand converts an addition instruction into a copy operation of the other input operand into the destination location. Also, if the other input operand is also the destination operand then the copy operation is redundant. For multiplications, a zero input operand results in a zero value of the destination operand regardless of the value of the other input operand. Thus, zero input operands can make some data dependencies and pipeline stages redundant for zero-optimizable and dependent instructions. Consequently, zero-optimizable and dependent instructions can be issued or commited earlier than normal or eliminated completely, as discussed below.

[2]Our optimizations also apply to loop optimizations (e.g., unrolling).

4.1.2. Early Instruction Issue/Commit. First, a zero-optimizable instruction can be issued once the zero operand is available if it makes other operands redundant. For example, I3 can be issued early becaue it is a multiplication and the zero value of R0 makes R2 redundant. Second, a zero-optimizable instruction could be committed early if the zero input determines its results and side effects. This is also the case for I3. Early issue and commit of zero-optimizable instructions can reduce pressure on processor resources and wait times of data dependent instructions, such as I4, since the dependencies are satisfied sooner.

4.1.3. Instruction Elimination. A zero-optimizable instruction can be squashed in the instruction queue if a zero input operand makes it an identity function and thus redundant. For this reason, I4 can be squashed since it is an addition and R2 is zero. Squashing an instruction can make the instructions that it depends on (producers) and those that depend on it (consumers) redundant, leading to more instruction squashing. For example, I5 becomes redundant (a silent store) and can be squashed, if I5 is squashed. Figure 5(c) shows the impact of squashing I4 and I5. We further observe that I1, I2, and I3 are now redundant in all but the last loop iteration, since their results (R2 and R4) are not used. We can squash these three instructions in all but the last iteration as shown in Figure 5(d). Compared to the orignal machine code sequence, the optimized code sequence will run much faster because of the squashed instructions, especially loads which often have high latency. Thus, by exploiting zero-optimizable instructions we can improve the performance of the inner loop of the gradient computation code.

4.2. Mechanisms

We propose extensions to a modern OOO pipeline for dynamic optimization of zero-optimizable instructions and the loops containing them. Our optimizer system comprises of lightweight extensions in the front-end and heavyweight extensions in the back-end of the pipeline. The split into lightweight and heavyweight components helps to avoid critical path delays in the front-end, while extracting maximum performance improvements. The front-end extensions enable early instruction issue/commit, while the back-end extensions enable instruction squashing in loops. Figure 6 presents

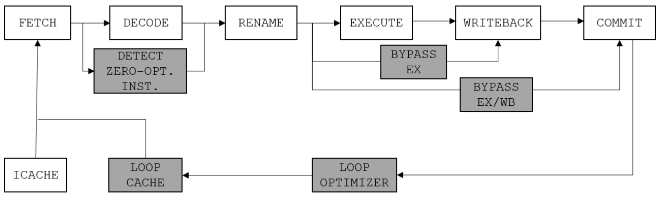


Figure 6: Overview of processor extensions.

an overview of augmenting a standard OOO six-stage pipeline with our optimization system.

4.2.1. Front-End Extensions. The objectives of the front-end extensions are the following: (i) detect zero-optimizable instructions in the instruction stream and (ii) bypass pipeline stages. These steps can be done in parallel with existing pipeline stages, as discussed below.

Detect Zero-optimizable Instructions. This is done in the decode stage by matching the instruction opcode against a predefined set of opcodes. Since the opcode set is small, i.e., addition and multiplication opcodes, the matching can be done in parallel to avoid extra delays. The zero-optimizable instructions detected here are marked for easy identification in later pipeline stages.

Bypass Pipeline Stages. We exploit zero operand values to bypass the execute and writeback stages whenever possible. This is done while a zero-optimizable instruction is in the instruction queue waiting for data dependencies. We extend the mechanism for detecting operand availability to also check whether or not the value is zero. For a multiplication instruction or a redundant addition instruction (i.e., destination operand is same as other source operand), we break outstanding data dependencies of the instruction making the instruction ready to issue. We extend the issue logic to allowing issuing instructions directly to stages later than the execute stage. This allows us to issue multiplication instructions to the writeback stage to zero the destination operand (and broadcast availability), and issue redundant additions to the commit stage.

4.2.2. Back-End Extensions. Our backend extensions consists of two components: (i) a mechanism for detecting and optimizing loops based on zero-optimizable instructions, and (ii) an optimized loop code cache. The goal is to eliminate instructions that are made redundant by zero inputs to zero-optimizable instructions in the loop. Thus, the optimized loop code is smaller than the original loop

Loop Optimizer. This mechanism scans the stream of committed instructions to detect loops that can be optimized based on zero-optimizable

Optimized Loop Code Cache. This mechanism caches the optimized code sequences generated by the loop optimizer and ensures the cached sequences are executed in place of the original code sequences.

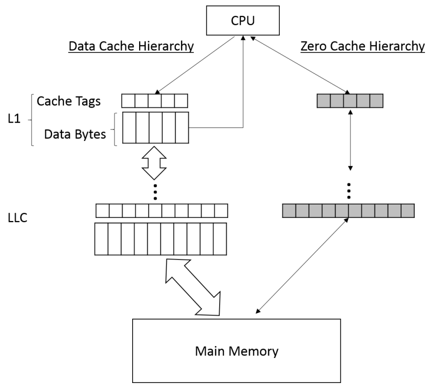


Figure 7: A Memory System with Zero Cache Hierarchy.

5. Cache Optimizations

Our cache optimizations for DNN training are based on the sparse nature of the performance critical data (e.g., activations, errors, etc.). 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 current design comprises of mechanisms for achieving the following: (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 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 is done promptly. We consider two alternatives for handling this: (i) an extra bit in the cache tags to identify zero cache lines, 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 bytes space of zero cache lines are unused. To avoid this waste, we adopt the second option in our current work.

5.2. Hierarchy of Zero Cache Lines

Figure 7 illustrates a memory system that is augmented with a cache hierarchy for zero cache lines, which we call

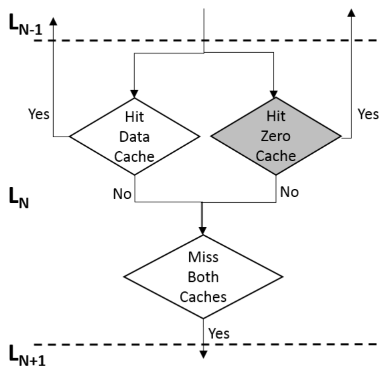


Figure 8: Handling read requests.

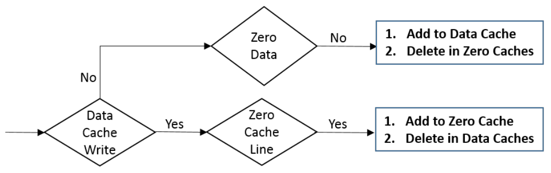


Figure 9: Handling processor writes.

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.

Data access requests from the processor are satisfied by accessing the two cache hierarchies in parallel to avoid introducing extra latency. Figure 8 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.

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 detector hardware [16] 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 9 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 along 3 dimensions: (i) the impact on single thread performance (6.2), (ii) the impact on multi-threading scalability in a server (6.3), and (iii) the impact on model parallelism performance (6.4).

6.1. Methodology

Image Recognition Task: Although we expect our optimizatons to be effective in general for training with gradient descent methods, we focus on image recognition because it represents an important class of AI problems for which significant accuracy improvements have being achieved through gradient descent training [29, 31, 14, 8, 20]. Specifically, we measure the impact of our optimizations on the training of high quality DNN models on 3 common image recognition workloads: (i) *MNIST* [33], (ii) *CIFAR-10* [28], and (iii) *Im-*

```
                                    …
for (i = 0; i < OutputNeuronCount; i++) {
  if (Error[i] != 0) {
     for (j = 0; j < InputNeuronCount; j++) {
         … += Error[i] * weights[i,j]
      }
   }
}
                                    …
```

Figure 10: Zero error signal optimization in back-propagation of a linear layer.

ageNet [15]. We describe each benchmark and correspodinging DNN in more details below.

- MNIST: The task is to classify 28x28 grayscale images of handwritten digits into 10 categories. The DNN is relatively small, containing about 2.5 million connections in 5 layers: 2 convolutional layers with pooling, 2 fully connected layers, and a 10-way output layer [8].
- CIFAR-10: The task is to classify 32x32 color images into 10 categories. The DNN is moderately-sized, containing about 28.5 million connections in 5 layers: 2 convolutional layers with pooling, 2 fully connected layers, and a 10-way output layer [29].
- ImageNet: The task is to classify 256x256 color images from a dataset of about 15 million images into a number of categories. There are 2 standard versions of this benchmark: (i) classifying 1.2 million images into 1000 categories (a.k.a., *ImageNet-1K*), and (ii) classifying the entire data set into 22000 categories (a.k.a. *ImageNet-22K*. We used the largest ImageNet task (i.e., *ImageNet-22K*), which is to classify 256x256 color images into $22,000$ categories. This DNN is extremely large, containing over 2 billion connections in 8 layers: 5 convolutional layers with pooling, 2 linear layers, and a $22,000$-way output layer [8].

Comparison to Software Approach: We compare our technique to a software approach that avoids zero-value computations without compact representations of sparse matrices or vectors, unlike CSR [23]. As discussed earlier, the dynamic nature of sparsity in training makes CSR less effective because the construction cost of the representation is incurred for each example. Rather, we test for zero values and skip the corresponding multiply-add operations. To derive the most benefit, this optimization is only applied when multiple multiply-add operations can be skipped for each zero value, such as during back-propagation or weight updates computation. The code snippet in Figure 10 illustrates this optimization for back-propagation of a linear layer, where computations involving an entire row of the weight matrix can be skipped for a zero error signal.

Simulation-based Approach: We conduct our performance experiments in a simulation enviroment, which allows us to easily prototype our proposed memory system extensions (Section **??**). We use a modified version of Memsim [1, 38], a multi-core simulator that models out-of-order cores coupled with a DDR3-1066 [25] DRAM simulator. All systems use a three-level cache hierarchy with a uniform 64B cache line size. We do not enforce inclusion in any level of the hierarchy. We use the state-of-the-art DRRIP cache replacement policy [**?**] for the last-level cache. All our evaluated systems use an aggressive multi-stream prefetcher [40] similar to the one implemented in IBM Power 6 [30].

6.2. Single Thread Performance

Here we show that our techniques improve single thread training performance by reducing the number of computations performed per training example.

6.3. Multi-threaded Performance

Here we show improved scalability by reducing the cache capacity and bandwidth pressure.

6.4. Model-parallelism Performance

Here we show improved scalability of model-parallelism because the reduced cache capacity and bandwidth pressure allows bigger models to fit on a machine.

7. Related Work

Our work is related to prior work on

- efficient sparse marix-vector computations [17, 23, 7, 27, 18, 39],
- reducing the cache [42, 43, 16, 24] and physical register file [26, 2] overheads of zero values.
- dynamic elimination of redundant computations [12, 9, 6, 37].

8. Conclusion

References

[1] Memsim. http://safari.ece.cmu.edu/tools.html, 2012.

[2] Saisanthosh Balakrishnan and Gurindar S. Sohi. Exploiting value locality in physical register files. In *MICRO*, 2003.

[3] Yoshua Bengio. Learning deep architectures for AI. *Foundations and Trends in Machine Learning*, 2009.

[4] Yoshua Bengio, Nicolas Boulanger-Lewandowski, and Razvan Pascanu. Advances in optimizing recurrent networks. In *Proc. ICASSP 38*, 2013.

[5] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *COMPSTAT*, 2010.

[6] J. Adam Butts and Guri Sohi. Dynamic dead-instruction detection and elimination. In *ASPLOS*, 2002.

[7] John B. Carter, Wilson C. Hsieh, Leigh Stoller, Mark R. Swanson, Lixin Zhang, Erik Brunvand, Al Davis, Chen-Chi Kuo, Ravindra Kuramkote, Michael A. Parker, Lambert Schaelicke, and Terry Tateyama. Impulse: Building a smarter memory controller. In *HPCA*, 1999.

[8] Trishul Chilimbi, Johnson Apacible, Karthik Kalyanaraman, and Yutaka Suzue. Project adam: Building an efficient and scalable deep learning training system. In *OSDI*, 2014.

[9] Jamison D. Collins, Dean M. Tullsen, Hong Wang, and John P. Shen. Dynamic speculative precomputation. In *MICRO*, 2001.

[10] Ronan Collobert and Jason Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *ICML*, 2008.

[11] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. Natural language processing (almost) from scratch. *The Journal of Machine Learning Research*, 2011.

[12] Daniel A. Connors, Hillery C. Hunter, Ben-Chung Cheng, and Wen-mei W. Hwu. Hardware support for dynamic activation of compiler-directed computation reuse. In *ASPLOS*, 2000.

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

[14] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew W. Senior, Paul A. Tucker, Ke Yang, and Andrew Y. Ng. Large scale distributed deep networks. In *NIPS*, 2012.

[15] 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.

[16] Julien Dusser, Thomas Piquet, and André Seznec. Zero-content augmented caches. In *ICS*, 2009.

[17] C. Stanley Eisenstat, MC Gursky, H. Martin Schultz, and H. Andrew Sherman. Yale sparse matrix package i: The symmetric codes. *International Journal for Numerical Methods in Engineering*, 1982.

[18] Jeremy Fowers, Kalin Ovtcharov, Karin Strauss, Eric Chung, and Greg Stitt. A high memory bandwidth fpga accelerator for sparse matrix-vector multiplication. In *International Symposium on Field-Programmable Custom Computing Machines*, 2014.

[19] Awni Hannun, Carl Case, Jared Casper, Bryan Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger, Sanjeev Satheesh, Shubho Sengupta, Adam Coates, et al. Deepspeech: Scaling up end-to-end speech recognition. *arXiv preprint arXiv:1412.5567*, 2014.

[20] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *CoRR*, 2015.

[21] Geoffrey Hinton, Li Deng, Dong Yu, George Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara Sainath, and Brian Kingsbury. Deep neural networks for acoustic modeling in speech recognition. *Signal Processing Magazine*, 2012.

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

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

[24] Mafijul Md. Islam and Per Stenstrom. Zero-value caches: Cancelling loads that return zero. In *PACT*, 2009.

[25] JEDEC. DDR3 SDRAM, JESD79-3F, 2012.

[26] Stephen Jourdan, Ronny Ronen, Michael Bekerman, Bishara Shomar, and Adi Yoaz. A novel renaming scheme to exploit value temporal locality through physical register reuse and unification.

[27] Srinidhi Kestur, John D. Davis, and Eric S. Chung. Towards a universal fpga matrix-vector multiplication architecture. In *FCCM*, 2012.

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

[29] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *NIPS*, 2012.

[30] H. Q. Le, W. J. Starke, J. S. Fields, D. Q. O'Connell, F. P. a nd Nguyen, B. J. Ronchetti, W. M. Sauer, E. M. Schwarz, and M. T. Vāden. Ibm power6 microarchitecture. *IBM JRD*, 51(6), 2007.

[31] Quoc Le, Marc'Aurelio Ranzato, Rajat Monga, Matthieu Devin, Kai Chen, Greg Corrado, Jeff Dean, and Andrew Ng. Building high-level features using large scale unsupervised learning. In *ICML*, 2012.

[32] Yann LeCun and Yoshua Bengio. The handbook of brain theory and neural networks. chapter Convolutional Networks for Images, Speech, and Time Series. 1998.

[33] Yann Lecun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, pages 2278–2324, 1998.

[34] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *NIPS*, 2013.

[35] Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann machines. In *ICML*, 2010.

[36] Andrew Y. Ng. Feature selection, l1 vs. l2 regularization, and rotational invariance. In *ICML*, 2004.

[37] Peter G. Sassone and D. Scott Wills. Dynamic strands: Collapsing speculative dependence chains for reducing pipeline communication. In *MICRO*, 2004.

[38] V. Seshadri, O. Mutlu, M. A. Kozuch, and T. C. Mowry. The Evicted-Address Filter: A Unified Mechanism to Address Both Cach e Pollution and Thrashing. In *PACT*, 2012.

[39] Vivek Seshadri, Gennady Pekhimenko, Olatunji Ruwase, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, Todd C. Mowry, and Trishul Chilimbi. Page overlays: An enhanced virtual memory framework to enable fine-grained memory management. In *ISCA*, 2015.

[40] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt. Feedback directed prefetching: Improving the performance and bandwidt h-efficiency of hardware prefetchers. In *HPCA*, 2007.

[41] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 2014.

[42] Luis Villa, Michael Zhang, and Krste Asanović. Dynamic zero compression for cache energy reduction. In *MICRO*, 2000.

[43] Youtao Zhang, Jun Yang, and Rajiv Gupta. Frequent value locality and value-centric data cache design. In *ASPLOS*, 2000.