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 [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 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.

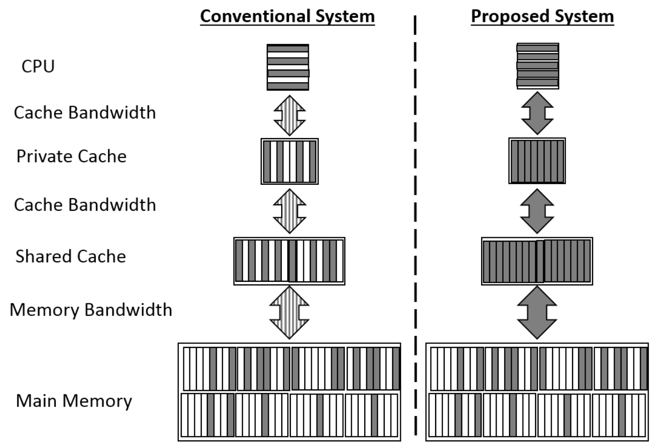

Figure 1: Processor and memory system utilization of sparse (white) and non-sparse (shaded) data.

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 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 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 processor and memory system techniques for reducing sparse data computation overheads. 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 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 in a simulation environment using real-world DNN training workloads for image recognition. The results show significant improvements in training time 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 [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 $l-1$:

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

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:

$$\delta_i = (true_i - a_i) \times f'(a_i) , \quad (2)$$

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 $l+1$:

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

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

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

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 data structures [20, 21, 1, 22, 23]. 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-forwad evaluation results 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) [21, 1] for faster convergence, and L_1 [20, 22] and Dropout [23] regularization methods for reducing overfitting. Trishul to help with this content

3.2 Sparsity in real-word image recognition

For a better 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 2 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 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 *4-Words* sparsity is the percentage of 4-wide vector multiply-adds that can be skipped because either the 4 activation values or 4 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, the impact of vectorization on computation sparsity varies across the training phases: no impact for backpropagation, modest sparsity reduction for weight updates, and significant sparsity reduction for feed-forward evalution (e.g., *4-Words* sparsity is half of *Word*). In summary, the results shows there is potential to signifcantly reduce the processing and 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. 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, cacheline granularity represents clusterings of sparse data values of size 16. Viewing data sparsity at both word and cacheline granularities helps to 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 can make the following three observations regarding data sparsity in DNN training from Figure 3. First, sparsity levels varies across the different data items: at word granularity, weights have negligible sparsity, 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. However, considerable levels of sparsity still remains at the cacheline granularity. In summary, the results show that cache capacity and bandwidth

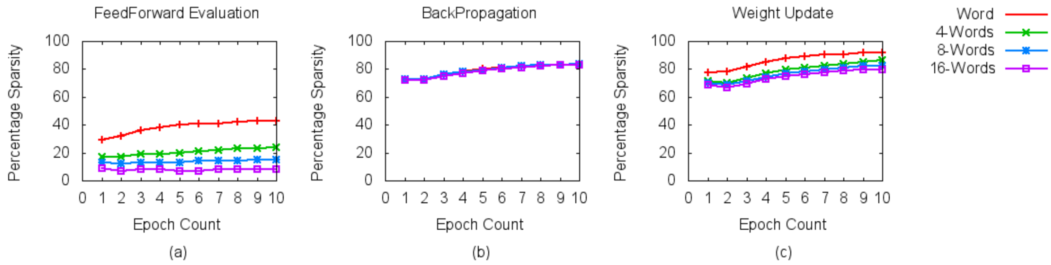

Figure 2: Computation sparsity in CIFAR-10 training.

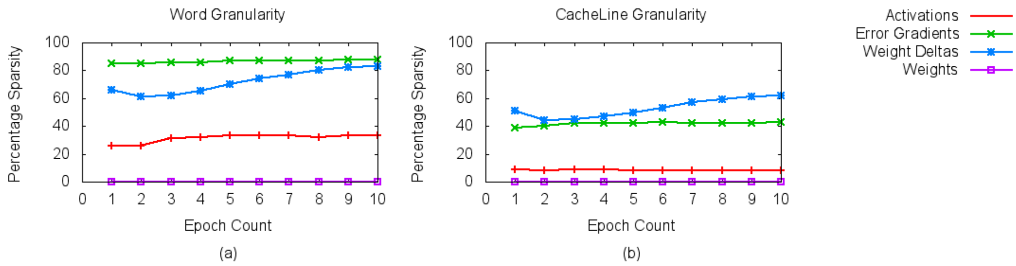

Figure 3: Data sparsity in CIFAR-10 training.

```
// 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.

requirements of DNN training can be significantly reduced by exploiting data sparsity, even at cacheline granularity.

3.3 Optimization Opportunities

We motivate our hardware optimizations for sparse data computations in training using a performance-critical kernel for computing weight deltas during 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 *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.

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 *activations[i]* and checking *errors[j]* have different performance impacts. Checking *errors[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 *activations[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 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 **??**), 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 of the loop[2], and includes five zero-

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

[2] Our techniques can improve loops that are optimized using standard compiler techniques (e.g., unrolling).

```
...
Loop:
I1    load    R2=[R3]
I2    load    R4=[R5]
I3    mul     R2=R2*R1
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)
```

```
...
Loop:
I1    load    R2=[R3]
I2    load    R4=[R5]
I3    mul     R2=R2*R1
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
...
(b)
```

```
...
I6    add     R3=R3+N*4
I7    add     R5=R5+N*4
I8    sub     R6=R6-N
I9    jne     Loop
...
(c)
```

```
...
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
...
(d)
```

Figure 5: (a) Deltas computation inner loop; optimized for R2 is zero in (b) one and (c) *N* iterations; (d) for R1 is zero.

optimizable instructions: I3, I4, I6, I7, and I8. R1 corresponds to the loop invariant *errors[j]*, the R2 target of I1 corresponds to *activations[i]*, and R4 target of I2 corresponds to *deltas[k]* of the code snippet in Figure 4.

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 our focus is on zero-optimizable instructions which perform additions or multiplications, we examine the effect of a zero input operand when the instructions have two input operands, and an input operand is also the output operand.

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.

4.1.2 Early Instruction Issue/Commit.

Normally, an instruction cannot be issued to the execution stage until both input operands are available. However, a multiplication instruction can be issued immediately an input operand is available as zero since the result (i.e., zero) is now independent of the other operand. For example, I3 can be issued once R1 is discovered to be zero without waiting for load I2 to complete; I3 becomes independent of I2. Moreover, I3 can proceed to the commit stage, bypassing the multiplication functional unit, because the result and side effect (set R2 to zero) can be directly generated. Early issue and commit of zero-optimizable instructions can reduce pipeline resource pressure and the waiting times of data dependent instructions, (e.g., I4), since those dependences are satisfied quicker.

4.1.3 Dead Instruction Elimination.

A zero input can make a zero-optimizable instruction redundant (or dead), and safe to eliminate from execution, as well as instructions which it depends on (producers) or which depend on it (consumers). We exploit this property to improve performance by executing optimized versions of loops, without dead instructions, when a zero-optimizable instruction input is zero. For concreteness, we examine the opportunities to eliminate dead instructions in loop 5(a) when either source operand of I3, R1 or R2, is zero.

A key challenge of dynamic detection of dead instructions is that the processor has limited view of future execution, and cannot guarantee, in general, that a value is not used again [25]. To address this issue, we exploit the fact that loops without control flow execute the same code sequence in each iteration, and instructions (values) that are dead across iterations can be identified. Thus, we eliminate instructions in all but the last loop iteration.

Intuitively, more instructions can be eliminated when R1 is zero compared to R2 because R1 is loop-invariant while R2 is not. So we first discuss optimizations when R2 is zero. Figure 5(a) shows that four instructions can be eliminated if R2 is zero. I4 is dead because it does not change R4, and the consumer I5 can execute correctly without it. I5 is dead because it is a silent store, while I2 is dead because R4 is dead at I5. Finally, I3 is dead because I4, its only consumer in the loop, is dead. I1 is live because the optimization depends on the value of R2. If we know apriori that R2 will be zero for*N* consecutive iterations then we can further improve performance by executing the non-loop code sequence in Figure 5(b) instead of *N* iterations of 5(a). We detect such opportunities when R2 is the first word of a zero cache line, then R2 will be zero for the next *N* iterations, where *N* is the minimum of R6 and 15. The processor is informed by our cache extensions when a load is serviced from a zero cache line.

If R1 is zero then I3 sets R2 to zero, which makes instructions I2 to I5 dead as discussed above. I1 is also dead since R1 does not depend on it. This is true for all but the last loop iteration since R1 is loop-invariant. Thus, we can replace the entire loop with the non-loop code sequence in Figure 5 to address the liveness concerns outside the loop.

4.2 Mechanisms

We propose extensions to a modern OOO pipeline for dynamic optimization of loops containing zero-optimizable instructions. 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

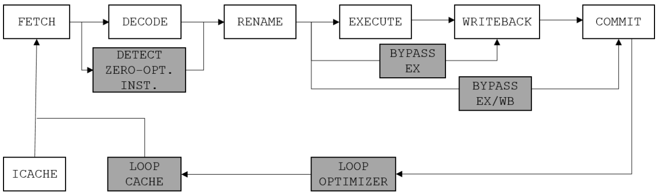

Figure 6: Overview of processor extensions.

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 dead instruction elimination in loops. Figure 6 presents 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 three mechanisms: (i) detecting loops that can benefit from our optimizations, (ii) generate and cache optimized code sequences to execute in place of loops when a zero-optimizable input is zero, and (iii) switch execution to optimized code sequences when appropriate.

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

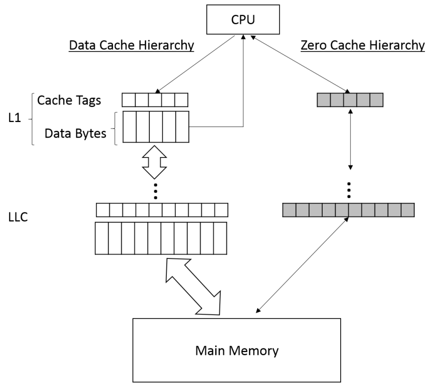

Figure 7: A Memory System with Zero Cache Hierarchy.

Loop Optimizer and Cache..
Optimize loop based on one zero-optimizable instruction input, and cache optimized code.

Optimized Code Executor..
Direct control flow to optimized code sequences in place of the original code sequences.

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

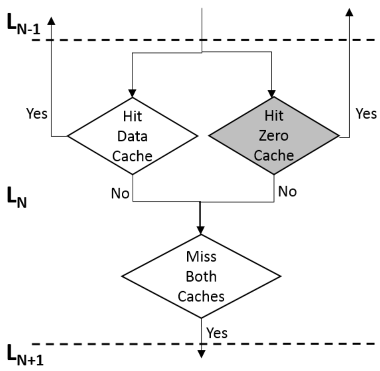

Figure 8: Handling read requests.

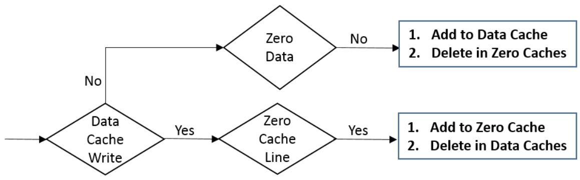

Figure 9: Handling processor writes.

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 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 Nth 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 [26] 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 [1, 2, 3, 4, 27]. Specifically, we measure the impact of our optimizations on the training of high quality DNN models on 3 common image recognition workloads: (i) *MNIST* [28], (ii) *CIFAR-10* [24], and (iii) *ImageNet* [29]. 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 [4].
- 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 [1].
- 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 [4].

Comparison to Software Approach:.

We compare our approach to a software optimization that dynamically checks training data for zeroes in order to skip multiply and addition operations (3.3). This software approach is more effective for training than using sparse representations of matrices and/or vectors [12] due to the dynamic nature of data sparsity in training. To mitigate the software check overheads, we apply this optimization only it situations that can yield significant benefits (e.g., skip inner loop execution).

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 [30, 31], a multi-core simulator that models out-of-order cores coupled with a DDR3-1066 [32] 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 [33] similar to the one implemented in IBM Power 6 [34].

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 [11, 12, 13, 14, 15, 35],
- reducing the cache [36, 37, 26, 38] and physical register file [39, 40] overheads of zero values.
- dynamic elimination of redundant computations [41, 42, 25, 43].

8. CONCLUSION

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 W. T.N., “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. A. Butts and G. Sohi, “Dynamic dead-instruction detection and elimination,” in *ASPLOS*, 2002.

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

[27] K. He, X. Zhang, S. Ren, and J. Sun, “Delving deep into rectifiers: Surpassing human-level performance on imagenet classification,” *CoRR*, 2015.

[28] 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.

[29] 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.

[30] “Memsim.” `http://safari.ece.cmu.edu/tools.html`, 2012.

[31] 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.

[32] JEDEC, “DDR3 SDRAM, JESD79-3F,” 2012.

[33] 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.

[34] 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. Vãden, “Ibm power6 microarchitecture,” *IBM JRD*, vol. 51, no. 6, 2007.

[35] 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.

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

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

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

[39] 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,”

[40] S. Balakrishnan and G. S. Sohi, “Exploiting value locality in physical register files,” in *MICRO*, 2003.

[41] 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.

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

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