Optimizing CNNs on Multicores for Scalability, Performance and Goodput

Samyam Rajbhandari
The Ohio State University
rajbhandari.4@osu.edu

Yuxiong He Olatunji Ruwase
Michael Carbin Trishul Chilimbi
Microsoft Research
yuxhe,olruwase,micarbin,trishulc@microsoft.com

Abstract

Convolutional Neural Networks (CNN) are a class of Artificial Neural Networks (ANN) that are highly efficient at the pattern recognition tasks that underlie difficult AI problems in a variety of domains, such as speech recognition, object recognition, and natural language processing. CNNs are, however, computationally intensive to train.

This paper presents the first characterization of the performance optimization opportunities for training CNNs on CPUs. Our characterization includes insights based on the structure of the network itself (i.e., intrinsic arithmetic intensity of the convolution and its scalability under parallelism) as well as dynamic properties of its execution (i.e., sparsity of the computation).

Given this characterization, we present an automatic framework called *spg*-CNN for optimizing CNN training on CPUs. It comprises of a computation scheduler for efficient parallel execution, and two code generators: one that optimizes for sparsity, and the other that optimizes for spatial reuse in convolutions.

We evaluate *spg*-CNN using convolutions from a variety of real world benchmarks, and show that *spg*-CNN can train CNNs faster than state-of-the-art approaches by an order of magnitude.

1. Introduction

Convolutional Neural Networks (CNN) are a class of Artificial Neural Networks (ANN) that are highly efficient at the pattern recognition tasks that underlie difficult AI problems in a variety of domains such as speech recognition, object recognition, and natural language processing [13, 15, 17, 32, 34, 38–41, 48, 49].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ASPLOS '17, April 08-12, 2017, Xi'an, China
© 2017 ACM. ISBN 978-1-4503-4465-4/17/04...$15.00
DOI: http://dx.doi.org/10.1145/3037697.3037745

CNNs are computationally intensive to train, especially for the most challenging tasks. For example, state-of-the-art CNNs [12, 14, 20] perform billions of floating-point operations on each training input, and must process millions of inputs to attain good task accuracies. The time to train a model is therefore on the order of days or weeks. Moreover, because CNNs are designed iteratively, it may require multiple months to reach a final model after multiple training iterations.

Our work focuses on developing an automatic framework for improving CNN performance on multi-core CPUs. The motivation for this focus (as opposed to a focus on GPUs or FPGAs) is twofold: 1) Modern multi-core CPUs are easily accessible and they can have more than a teraflop of peak performance (Eg. Xeon Processor E7-8895 v3), which is enough to train medium and moderately large CNNs in reasonable time. 2) There is an abundance of multi-core CPU clusters — which are used in state-of-the-art large CNNs [12, 20] — that are easily accessible to both academic community and industry. Performance improvements on CPUs therefore simultaneously benefits many low-, mid-, and high-end users.

1.1 Convolutional Neural Networks

In a traditional ANN, the input for a recognition problem is represented as an array of floating-point values (e.g., pixels for image classification) and passed into a stack of *layers*. Each layer computes a non-linear function of either the problem's input or the output of the previous layer. The multi-dimensional output of a layer is composed of *neurons*, where each neuron first computes a linear function (parameterized by a set of coefficients or *weights*) of all the outputs of the previous layer. The neuron then applies a non-linear function to that result.

A CNN differs from a traditional ANN in that each neuron connects to only a small region of the previous layer. Moreover, the neurons of a layer in a CNN are grouped into *feature maps*, in which all neurons in a feature map apply the exact same function (or *feature*) to its corresponding input region.

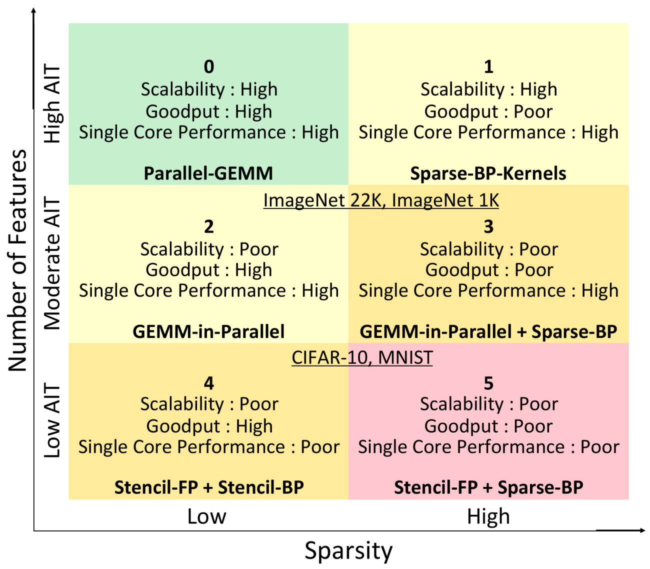


Figure 1: CNN performance characteristics of using Unfold+Parallel-GEMM as a function of AIT ($\approx 2\times$ Number of features) and sparsity with respect to scalability, goodput and single core performance. The design space is divided into six regions. The figure shows their different performance characteristics and the optimization techniques of our work (depicted in bold) for improving them. In addition, it illustrates regions that real world image classification benchmarks (ImageNet-22k, ImageNet-1K, CIFAR-10, MNIST) occupy in the convolution space.

For CNN training on multi-core CPUs, state-of-the-art CNN infrastructures such as CAFFE [33], Theano [7], Torch7 [16], Chainer [1], CNTK [2], and TensorFlow [5] all implement convolution computation using a process we refer to as *Unfold+Parallel-GEMM* [8], where inputs are unfolded to cast it as a matrix-multiply (MM). The resulting MM is computed efficiently by linking to third party Basic Linear Algebra Subprogram Libraries (BLAS) such as MKL [30], ATLAS [53] or OpenBLAS [54], which offer highly optimized Parallel-GEMM (General Matrix Multiply) implementations for multi-core CPUs.

1.2 Performance Characterization

Fig. 1 presents a characterization of the performance of Unfold+Parallel-GEMM as a function of the *arithmetic intensity* and *sparsity* of the CNN's computation.

- **Arithmetic Intensity (AIT)** — the ratio of number of arithmetic operations to the number of memory operations in a computation. A high AIT is necessary to get high performance because memory operations are slower than arithmetic operations.
- **Sparsity** — the fraction of elements in a data array that are zeros. High sparsity in the data means that a naive execution approach performs many computationally intensive operations that could instead be elided without affecting the computation's correctness.

Based upon the AIT and Sparsity of a convolution computation, we define three performance characteristics of Unfold+Parallel-GEM: *scalability*, *single core performance*, and *goodput*.

Scalability: Unfold+Parallel-GEMM scales poorly when the MM corresponding to a convolution computation does not have high AIT to begin with (Fig. 1, Region 2, 3, 4 and 5). We show that as we increase the number of cores, the number of arithmetic operations in Parallel-GEMM per core reduces proportionately, while the number of memory operation does not, resulting in a reduced AIT per core.

Single Core Performance: While large convolutions have adequate AIT to achieve high single core performance using Unfold+Parallel-GEMM (Fig. 1, Region 0, 1, 2 and 3), small and medium convolutions exhibit poor single core performance due to very low AIT (Fig. 1, Region 4 and 5). We show that the unfolding increases the number of memory operations, and thus reduces AIT.

Goodput: We define *goodput* as the rate of useful work in a computation. For example, it is possible to simultaneously achieve high throughput and low goodput if most of the computation is avoidable. Because CNN computations often involve sparse data, Unfold+Parallel-GEMM's dense MM results in poor goodput. (Fig. 1, Region 1, 3 and 5).

1.3 Optimization Framework for CNNs

We present *spg*-CNN, an optimization framework for CNNs that achieves high Scalability, Performance and Goodput. *spg*-CNN consists of three key components that work collectively to address the three limitations of Unfold+Parallel-GEMM.

- **GEMM-in-Parallel:** *spg*-CNN improves scalability over Parallel-GEMM by running multiple instances of single-threaded GEMM in parallel (Fig. 1 Region 2, and 3). This simple re-scheduling prevents AIT reduction when scaling to multiple cores.
- **Stencil-Kernel:** *spg*-CNN improves single-core performance over Unfold+Parallel+GEMM for small convolutions where unfolding results in very low AIT. We develop a new approach inspired by stencil computations [19], which computes CNN through direct convolution without unfolding, exploits spatial reuse of inputs and improves AIT (Fig. 1 Region 4 and 5).
- **Sparse-Kernel**: *spg*-CNN improves the goodput of sparse CNN computations over Unfold+Parallel-GEMM and its sparse variants [30] (Fig. 1 Region 1, 3 and 5). Sparse GEMM libraries are effective when both input matrices are highly ($> 95\%$) sparse [43], but the inputs of CNN computations are typically a moderately ($50-95\%$) sparse matrix and a dense one. *spg*-CNN incorporates a *pointer shifting* technique to compose a sparse convolution as a

series of small and dense MMs, performing computation in place without unfolding. Additionally, it generates efficient SIMD instructions, and exploits locality enhancing sparse data structures, to achieve high goodput on sparse and dense inputs.

Given a CNN, *spg*-CNN generates codes and chooses the fastest among Parallel-GEMM, GEMM-in-Parallel, Sparse-Kernel and Stencil-Kernel for the forward-propagation (FP) and back-propagation (BP) phases of each layer, optimizing the training time. For CNNs with different network structures, *spg*-CNN generates the best configurations based on their performance characteristics.

1.4 Contributions

This paper makes the following contributions:

- **Performance Characterization.** We systematically analyze the CNN performance of using Unfold+Parallel-GEMM with respect to scalability, single-core performance and goodput in a clear design space based on AIT and sparsity, show its limitations and identify the root causes (Section 3).
- **Optimization Framework.** We develop *spg*-CNN, an optimization framework for CNNs, which introduces and integrates three key components, i) an alternate schedule using GEMM-in-Parallel, improving scalability, ii) a stencil-based code generator, improving per-core performance, and iii) a sparse code generator, exploiting sparsity and improving goodput, into a unified code generator that produces optimized codes for various CNN computations (Section 4).
- **Evaluation:** We evaluate *spg*-CNN using the CNN training frameworks Adam [12] and Caffe [33]. The results show performance improvements of up to 16x on convolutions from real-world benchmarks such as ImageNet-22K, ImageNet-1K, CIFAR-10 and MNIST. The results also show an end-to-end speed up of 8.3x on CIFAR-10 image recognition benchmark (Section 5).

2. Background

In this section we present the basic elements of Artificial Neural Networks, Convolution Neural Networks, and the contemporary methods for training these networks.

2.1 Artificial Neural Networks (ANN)

An ANN comprises of a stack of *layers* each of which is a group of *neurons.*

Layers and Neurons: A layer, denoted by A_{l+1}, is a group of neurons in which each neuron in the layer computes a non-linear function of the outputs of neurons in the preceding (lower) layer. A neuron $A_{l+1}[i]$ specifically applies a non-linear function (ϕ) to a linear combination of the outputs of the layer below (A_l) using a set of *weights* (W).

$$A_{l+1}[i] = \phi\left(\sum_j A_l[j] \times W[i,j]\right) \quad (1)$$

Forward Propagation (FP): FP evaluates an entire network to compute the network's result on an input. It computes the network's result by successively computing the output activations of the neurons of each layer based on the previous layer activations.

Backward Propagation (BP): BP computes the *error gradients* of a network's weights with respect to a loss function It works backwards from the output layer, using the chain rule to compute the error gradient of each weight in a lower layer as a function of the upper layers.

Stochastic Gradient Descent: ANNs are often trained using *stochastic gradient descent* (SGD). For an input example and corresponding target output, stochastic gradient descent executes FP to compute the network's output and then executes BP to compute error gradients of the weights. SGD then multiplies the error gradient of each weight by the value of the input connected to that weight to produce the *delta weights*. The delta weights are the updates that SGD applies to the existing weights to produce an updated model. Training then proceeds by repeating the procedure on a new input example with the updated model.

2.2 Convolutional Neural Network (CNN)

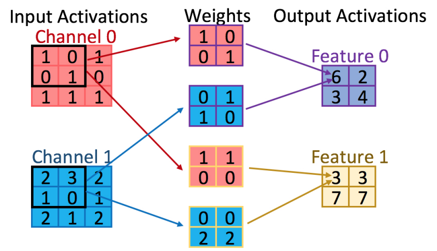
A CNN is a sub-class of ANNs where neurons in a layer are only connected to neurons in its local surroundings in the previous layer, and the weights are shared.

Layers and Neurons: Fig. 2a shows an example of a convolution on a two dimensional image of size 3×3 with two input features (Channel 0 and Channel 1), corresponding to, for example, the red and blue channels of the image. The convolution has two output features, Feature 0 and Feature 1, Each feature has individual sets of weights that correspond to each input feature. The weights for the first feature are the top two matrices under the Weights column whereas the weights for the second feature are the bottom two matrices. The kernel size of each output feature is 2×2, which is the size of each weight matrix.

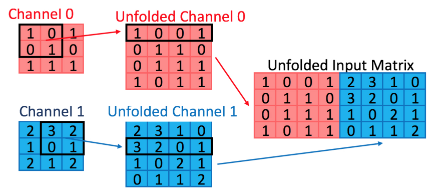
To produce the first element of Feature 0, the convolution computes the inner product of the sub-region of Channel 0 within the black boundary and the feature's weights that correspond to that channel. The convolution then sums this result with the inner product of the sub-region of Channel 1 within the black boundary and the feature's weights that correspond to that channel.

FP: A fully general convolution operation in two dimensions is represented using a convolution kernel of 5-tuple $\langle N_f, F_y, F_x, s_y, s_x\rangle$. This convolution operation can be mathematically written as

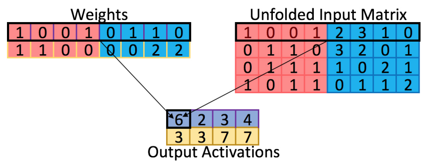
$$O[f,y,x] = \sum_{c,k_y,k_x=0}^{N_c,F_y,F_x} I[c, y{\times}s_y{+}k_y, x{\times}s_x{+}k_x] \times W[f,c,k_y,k_x] \quad (2)$$



(a) 2D Convolution



(b) Unfolding



(c) MM $O = W \cdot U^T$

Figure 2: (a) An example of a 2D convolution using a 2×2 kernel with 2 input features of size 3×3 and 2 output features. (b) Unfolding a 2D image with two channels for convolving with a 2×2 convolution kernel. (c) MM $O = W \cdot U^T$ representing a 2D convolution using a 2×2 kernel with 2 output features.

where O and I represent the output and input activations, y and x are the spatial coordinates of the output activation, f represents the features of the output activations, c represents the features of the input activations, s_y and s_x are the strides along y and x, and k_y and k_x represents the kernel coordinates (weights corresponding to connections that are a distance of k_y and k_x from the output neuron along y and x dimensions). We use N_f, N_c, F_y and F_x to represent the number of output features, number of input features, kernel width along y dimension and kernel width along x dimension, respectively.

BP and SGD: Just as with ANNs, SGD updates the network weights using the error gradients computed by BP. For a given layer, Eq. 3 computes the error in the input activations (E_I) based on the errors of the output activations (E_O), and Eq. 4 computes the corrections to the weights (dW) using the input activations I and the error in the output activations (E_O). N_y and N_x represent the spatial size of the output activations along y and x dimensions.

$$E_I[c,y,x] = \sum_{f,k_y,k_x=0}^{N_f,F_y,F_x} E_O[f, \frac{y-k_y}{s_y}, \frac{x-k_x}{s_x}] \times W[f,c,k_y,k_x] \tag{3}$$

$$dW[f,c,k_y,k_x] = \sum_{y,x=0}^{N_y,N_x} E_O[f,y,x] \times I[c, y \times s_y + k_y, x \times s_x + k_x] \tag{4}$$

2.3 Executing CNNs

The state-of-the-art execution method for CNNs is to use a two-step process [8] that we term Unfold+Parallel-GEMM. We describe this method in FP as below, and BP calculations are done with similar transformation but in the reverse order.

(1) Unfold: The first step is to unfold the input activation vector into a matrix. Fig. 2b illustrates the input activation unfolding procedure for the convolution presented in Fig. 2a. For each input channel, the unfolding procedure flattens the inputs to each kernel application into a row vector. The sequential concatenation of each row vector produces the unfolded representation of a channel. In the last step of unfolding, the procedure stacks each unfolded input channel from left to right to produce the final unfolded input matrix.

(2) Matrix Multiply: The second step performs a matrix-matrix multiplication, with one matrix consisting of the layer's weights and the other consisting of the unfolded activations, as shown in Fig. 2c. It constructs the weight matrix by stacking row vectors that correspond to the flattened representation of the weights for each feature. For example, the black box of the matrix titled "Weights" in the figure highlights the flattened representation of the weights of the first feature (which includes weights for both Channel 0 and Channel 1).

3. Performance Characterization

We categorize CNNs based on arithmetic intensity (AIT) and sparsity, and characterize the performance of using Unfold+Parallel-GEMM in terms of 1) *single-core performance*, 2) *multi-core scalability* and 3) *goodput*. This study exposes a range of performance issues of the current approach and guides the design of *spg*-CNN. We produced all of the experimental results presented in this section on an Intel(R) Xeon(R) CPU E5-2650 with 16 cores with a peak performance of 41.6GFlops per core. For MM, we used the OpenBLAS library.

3.1 Single-Core Performance

Unfold+Parallel-GEMM's unfolding procedure can reduce the maximum achievable fraction of the intrinsic AIT of the convolution operation, therefore resulting in poor single core performance (Fig. 1 Region 4 and 5). The AIT of a convolution is $\frac{|A|}{|I|+|W|+|O|}$, where $|A|$ is the number of arithmetic operations and $|I|+|W|+|O|$ is the number of memory accesses. The sets I, W, and O correspond to the input activations, weights, and output activations, respectively. Their sizes are calculated as follows:

$$|A| = 2N_f N_x N_y N_c F_y F_x \quad (5)$$

$$|I| = N_x N_y N_c \quad (6)$$

$$|W| = N_f F_x F_y N_c \quad (7)$$

$$|O| = N_f(N_x - F_x + 1)(N_y - F_y + 1)\ . \quad (8)$$

The unfolding procedure increases the size of the activations by approximately a factor of $F_x F_y$. In addition, the unfolded inputs need to be stored before the MM, doubling the number of memory access to the unfolded input. Therefore, the minimum number of memory accesses of Unfold+Parallel-GEMM is $2|U| + |W| + |O|$, where U is the unfolded inputs with size

$$|U| = (N_x - F_x + 1)(N_y - F_y + 1)N_c F_x F_y\ .$$

The resulting fraction of the intrinsic AIT of convolution that Unfold Parallel-GEMM can achieve is at most r, where

$$r = \frac{|I| + |W| + |O|}{2|U| + |W| + |O|}\ .$$

There are two dominant axes on which the fraction of intrinsic AIT achieved via Unfold+Parallel-GEMM changes:

- **Kernel Size:** When convolution kernel size is much smaller than the input dimensions (with $F_x \ll N_x$ and $F_y \ll N_y$), increase in the kernel size reduces r. However, as the size of the convolution kernels approaches the size of the input dimensions (with $F_x = N_x$ and $F_y = N_y$ at the limit), the convolution itself more closely resembles a matrix multiply ($r \approx 1$).
- **Output Feature Count:** As output feature count (N_f) increases, the number of memory accesses in the convolution are dominated by those to the weights ($r \approx 1$).

Table. 1 illustrates the relationship between these dimensions and the AIT of Unfold+Parallel-GEMM. Specifically, Unfold+Parallel-GEMM achieves a higher fraction of the intrinsic AIT of the convolution for larger kernel sizes and larger output feature counts because either little unfolding is required (larger kernel size) or the overhead of unfolding is diminished by the size of the weights (output feature count). However, small CNNs suffer from poor single-core performance due to very low AITs caused by the unfolding process.

3.2 Multicore Scalability

On shared-memory multicore systems, large MM achieves near peak performance, and scales perfectly to all cores using Parallel-GEMM (Fig. 1 Region 0 and 1). However, there is a subspace of MM that renders sub-optimal performance, or scaling, or both (Fig. 1 Region 2, 3, 4 and 5), because of low AIT per core.

Fig. 3a presents the absolute performance per core and scalability for Parallel-GEMM corresponding to different convolution operations. We timed the execution of three MM corresponding to FP, gradient calculations and delta-weight calculations and calculated the GFlops per core. Each curve in the plot shows a different convolution.

ID	$N_x(=N_y), N_f, N_c, F_x(=F_y)$	Intrinsic AIT	Unfold+GEMM	Region (Reg)
0	32,32,32,4	362	25	4,5
1	64,1024,512,2	2015	725	0,1
2	256,256,128,3	1510	226	2,3
3	128,128,64,7	3561	113	2,3
4	128,512,256,5	6567	456	2,3
5	64,64,16,11	1921	44	4,5

Table 1: Different Convolutions, their intrinsic AIT, the AIT corresponding to Unfold+GEMM, and the region in Fig. 1 that they belong to. These benchmarks were chosen to represent convolutions with high, moderate and low AIT, arching over a full spectrum of convolutions spanned by kernel size and number of features. For varying levels of sparsity, these six benchmarks cover the entire performance characterization space shown in Fig. 1.

Fig. 3a shows that Parallel-GEMM does not scale linearly in the number of cores. The AIT of MM depends on the size of the matrix. Specifically, MM of two square matrices of size $n \times n$ has $2n^3$ arithmetic operations and at least $3n^2$ load + store operations. MM therefore has an AIT of at most $\frac{2n}{3}$. While scaling Parallel-GEMM to multiple cores evenly divides the total number of operations of the MM across the cores, it does not do the same for loads to and stores from per core private memory. This reduces the AIT per core.

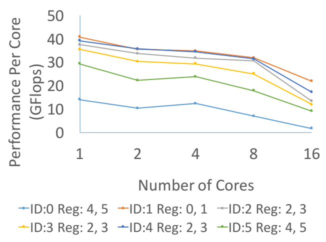
AIT per Core : This is calculated by dividing the total amount of computation done per core by the number of memory load and stores to its private memory (L2 cache).

For instance consider a square MM where two square input matrices A and B of size $n \times n$ are multiplied together to produce matrix C also of size $n \times n$. The AIT of this MM is $\frac{2n}{3}$. On a dual-core machine, each core can compute half of output matrix C. Computing half of C requires either half of A and entire B, or entire A and half of B, depending on whether C is partitioned along row, or column. Therefore, the AIT per core is $\frac{0.5 \times 2n \times n \times n}{0.5n \times n + n \times n + 0.5n \times n} = \frac{n}{2}$ resulting in a reduction from $\frac{2n}{3}$. Scaling to larger number of cores further reduces the AIT per core. Thus, we see poor scalability in Fig. 3a as we increase the number of threads.

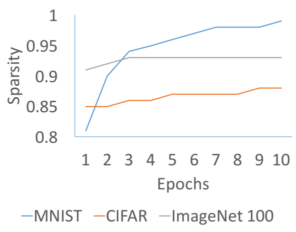
3.3 Goodput

The conventional approach to BP uses dense MM to compute error activations and delta-weight updates. One of the inputs to both of these calculations is the output activation error, denoted as E_O in Eq. 3 and Eq. 4. E_O represents corrections to the original activations of the layer. In many neural network models, these corrections tend to be sparse, with few non-zero elements. When a network exhibits high sparsity, dense MM is inefficient because a significant portion of the computation is multiplying and adding zero values.

In the presence of sparsity, we define efficiency in terms of goodput. *Goodput* is the rate of doing non-zero compu-



(a) Parallel-GEMM Scalability



(b) Sparsity across Epochs

Figure 3: (a) Scalability of Parallel-GEMM on up to 16 cores for convolutions shown in Table. 1. (b) Sparsity across multiple epochs for three CNN benchmarks.

tation (Eq. 9). By assuming that only the output activation error in BP calculation is sparse, we establish an upper limit on the goodput of Parallel-GEMM(Eq. 10), which we refer to as the goodput of Parallel-GEMM based BP.

$$\begin{aligned} Goodput &= \frac{Number\ of\ NonZero\ Flops}{TimeElapsed} && (9) \\ &\leq (1.0 - Sparsity) \times Throughput\ . && (10) \end{aligned}$$

For BP of real world DNN benchmarks, the goodput of Parallel-GEMM is significantly smaller than throughput because there is high level of sparsity in BP. Fig.3b shows sparsity in activation errors across multiple training epochs for three real world benchmarks, MNIST, CIFAR and ImageNet-100. After the second epoch, all three benchmarks have a sparsity level of more than 85% in their activation errors. Further, as the model becomes more accurate, these activation errors become even sparser.

At this level of sparsity, dense GEMM goodput is at most 15% of the throughput. For example, if dense GEMM has a throughput of 60 GFops per core, then the goodput is only 9 GFlops, while the remaining 51 Gflops is wasted in zero calculations. In Fig. 1, regions 1, 3 and 5 have poor goodput.

4. Optimization Framework

This section presents *spg*-CNN, a CNN training optimization framework. Motivated by Sec. 3, *spg*-CNN introduces and integrates three techniques specified below into a unified optimization framework, generating optimized codes covering various CNN computations with different characteristics. The CNN description to *spg*-CNN can be specified using Google Protocol Buffer [3] similar to how CAFFE [33] describes its inputs.

Stencil-Kernel: Improve single-core performance for convolutions with smaller kernel sizes and fewer output features through custom-generated stencil-based kernels for FP. These kernels avoid unfolding and therefore avoid the AIT reduction that unfolding introduces.

GEMM-in-Parallel: Improve multicore scalability by running single-threaded GEMM on multiple training inputs in parallel. This technique avoids the AIT reduction that arises from partitioning the computation via Parallel-GEMM.

Sparse-Kernel: Improve the goodput for sparse models through custom-generated sparse-based kernels for BP. These kernels elide computations on zero values that have no effect on the convolution's result.

We next describe each of our techniques in the order of their relative complexity to facilitate readability.

4.1 GEMM-in-Parallel

We improve the scalability of convolutions on multi-core hardware using GEMM-in-Parallel: multiple instances of single-threaded GEMM run concurrently on different cores. GEMM-in-Parallel performs better than Parallel-GEMM because inputs are not divided across cores, and so, per-core AIT (and therefore performance) stays the same.

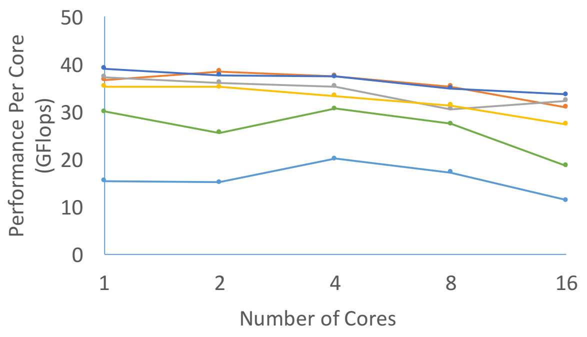
Evaluation: Fig.4a shows the scalability of GEMM-in-Parallel, depicting the absolute performance per core with varying core number from 1 to 16. The results show the performance per core is roughly steady, and drops by $< 15\%$ on average. In contrast, the average performance drop per core for Parallel-GEMM is $> 50\%$ (Fig.3a). Fig.4b shows the relative speedup of GEMM-in-Parallel over Parallel-GEMM. The relative speedup grows with more cores, indicating better scalability. Also, convolutions with fewer output features benefit more from GEMM-in-Parallel because their low AIT is further reduced by Parallel-GEMM on more cores.

4.2 Sparse-Kernel

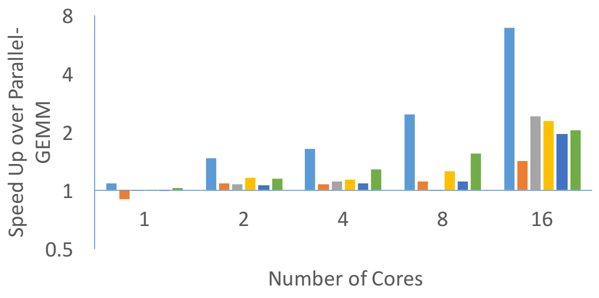
Our framework takes advantage of the sparsity in error gradients to improve BP goodput by using sparse kernels to compute error gradients and weight deltas. These kernels exploit efficient data representation, vectorization, cache and TLB optimization as well as a new pointer shifting technique, to achieve high goodput on sparse and dense inputs. Sparse convolution is performed in place, without unfolding, as a composition of small and dense MMs.

Sparse Data Representation: Error gradients are stored in Column Tiled-Compressed Sparse Row (CT-CSR), our adaption of the popular CSR format for sparse matrices. In CSR, a sparse matrix is stored using three arrays: i) value array for storing non-zero values, ii) column index array for storing column indices of the non-zero values, and iii) row index array for storing the starting position of each row in the data array or column index array. In CT-CSR, the sparse matrix is tiled along columns, and each tile is stored in CSR (see Fig. 5a). CT-CSR provides better locality than CSR by tiling along both row and column of the sparse matrix to enable greater reuse of tile elements (in both dimensions).

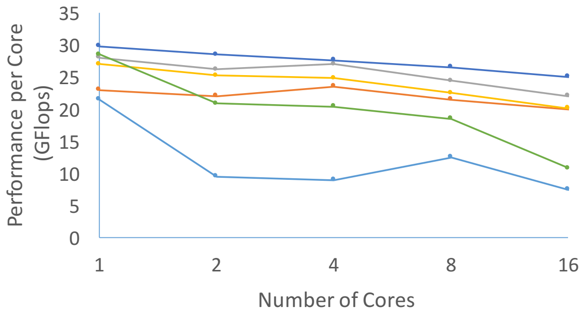
Another advantage of CT-CSR is the reduction in TLB misses when accessing elements within a tile. In CT-CSR elements of two adjacent rows within a tile are also adjacent in memory. Without this explicit tiling, elements corresponding to two adjacent rows may be far apart depending on the column width of the entire matrix requiring two TLB



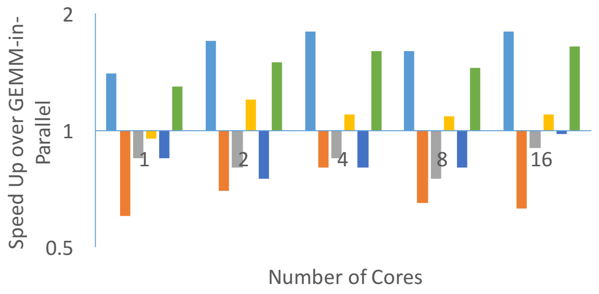
(a) Scalability of GEMM-in-Parallel on up to 16 cores.



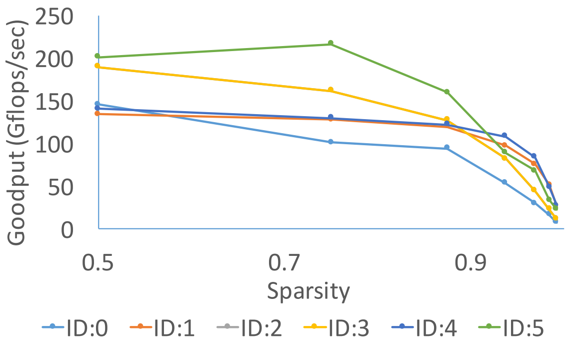
(b) Relative Speed Up of GEMM-in-Parallel over Parallel-GEMM



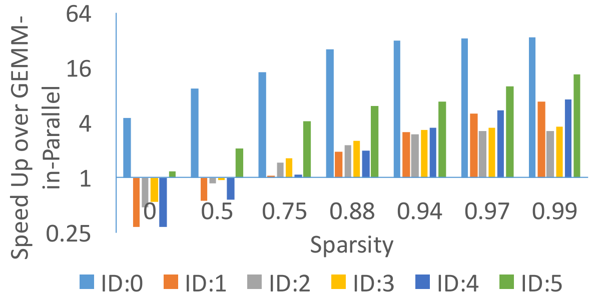
(c) Scalability and Performance of Stencil-Kernel (FP).



(d) Speed up of Stencil-Kernel (FP) over GEMM-in-Parallel.



(e) Goodput of Sparse-Kernel (BP) as a function of sparsity.



(f) Speed up from Sparse-Kernel (BP) over GEMM-in-Parallel

Figure 4: The set of figures shows absolute performance for six dense convolutions, goodput for six sparse convolutions, and relative speedups over baselines achieved using the three optimization techniques in *spg*-CNN. Each convolution is labeled using an ID in the legend. The convolution specification and the Region that it belongs to can be identified by looking up the ID in Table. 1. Fig. a and b indicate that GEMM-in-Parallel significantly improves scalability in Region 2 (ID: 2, 3 and 4). Fig. c and d show that Stencil-Kernel (FP) significantly improves performance in Region 4 (ID: 0 and 5). Fig. e and f show significant goodput improvement in Regions 1 (ID: 1), 3 (ID: 2, 3 and 4), and 5 (ID: 0 and 5).

lines to access them. Thus, CT-CSR reduces the number of TLB entries required to hold the tile in cache, resulting in a reduction in TLB misses.

spg-CNN generates efficient AVX vector instructions using Intel Intrinsics, that vectorize across the dense input, optimize for cache locality, and reduce TLB misses. Our locality optimization techniques are similar to [26].

Vectorization: Our code generator uses sparse-dense matrix multiplication as a basic code block for the generated code to efficient execute the convolution with vectorization and without (un)folding. The output of these basic code blocks are computed in place without unfolding using a novel technique that we call *pointer shifting*.

For illustration, consider the error gradient calculation in Eq. 3. First we identify the MM operations within this calculation. We rewrite the equation as Eq. 11, where $S[c, y, x, k_y, k_x]$ is given by Eq. 12. For a fixed value of

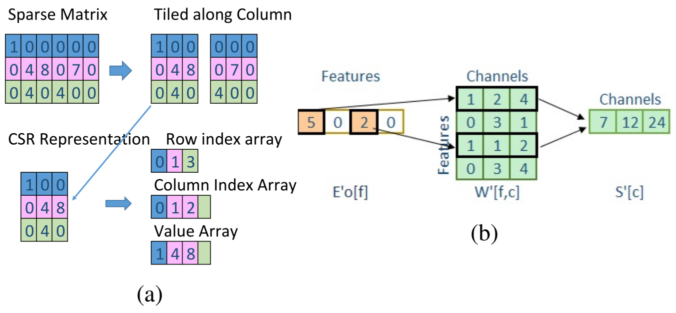


Figure 5: (a) Sparse matrix is first tiled along the columns, then each tile is stored in CT-CSR format. (b) A sparse column matrix is multiplied with a dense matrix to produce a dense column matrix as showing in Eq. 13.

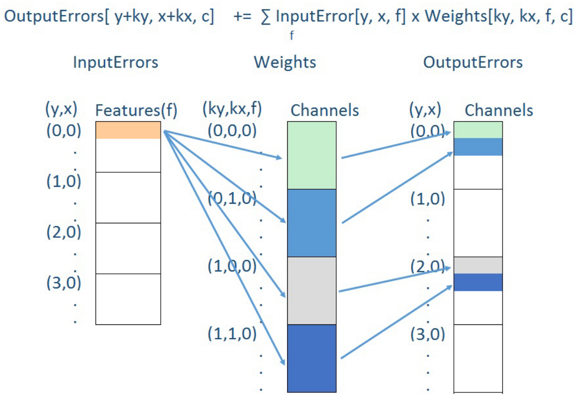


Figure 6: Sparse Kernel for backward gradient calculations. Each arrow on the left represents a sparse-dense MM. The arrows on the right show the result's storage location.

k_y, k_x, y and x, Eq. 12 reduces to Eq. 13, which is a matrix-matrix multiply.

$$E_I[c,y,x] = \sum_{k_y,k_x=0}^{F_y,F_x} S[c,y,x,k_y,k_x] \tag{11}$$

$$S[c,k_y,k_x] = \sum_{f}^{N_f} E_O[f, \frac{y-k_y}{s_y}, \frac{x-k_x}{s_x}] \times W[f,c,k_y,k_x] \tag{12}$$

$$S'[c] = \sum_{f}^{N_f} E'_O[f] \times W'[f,c]\,. \tag{13}$$

Because E'_O is sparse, and W' is dense, Eq. 13 can be computed efficiently by vectorizing along channels (c), as shown in Fig. 5b. We first perform data layout transformation: the weights (W') and outputs (E_I or S') are transformed so that c is the fastest varying dimension in memory, and input (E_O or E'_O) is transformed so that f is the fastest varying dimension in memory. Then each non-zero element $E'_O[f]$ is multiplied with the corresponding vector $W'[f,*]$, shown using the bold black boxes to produce a vector of $S'[*]$, where $*$ represents $c = 0,1,2$ stored as a vector.

Now, consider all the inputs $E_O[y',x',f]$ contributing to output vector $E_I[y,x,*]$. This can be written as

$$E_I[y,x,*] \leftarrow E_O[f, \frac{y-k_y}{s_y}, \frac{x-k_x}{s_x}] \tag{14}$$

where $y' = \frac{y-k_y}{s_y}$ and $x' = \frac{x-k_x}{s_x}$ for a given value of k_y and k_x. In other words, each input value E_O contributes to multiple output vectors E_I, given by

$$E_O[y',x',f] \rightarrow E_I[y's_y + k_y, x's_x + k_x, *]\,. \tag{15}$$

Using this relation, we can identify the position of the output vector $E_I[y,x,*]$ for a given input $E_O[f,y',x']$, and kernel coordinates k_y and k_x. We can thus compute sparse MM given by Eq. 13 in place, for all values of k_y and k_x, without unrolling them. This is shown in Fig. 6. Each arrow between E_O and W represents a sparse MM between input $E[y',x',f]$, weights $W[k_y,k_x,f,*]$, for different values of k_y and k_x. The arrows between W and E_I shows the position of the output vector resulting from the sparse MM.

Evaluation: We refer to the code generated by our sparse code generator as Sparse-Kernel (BP). Fig. 4e shows its goodput for various levels of sparsity on 16 cores. The costs of data-layout transformations and creating the CT-CSR representation are included. For $< 90\%$ sparsity, our kernels achieve consistently high goodput for all sizes of the convolutions. The performance drop after $> 90\%$ sparsity is due to the bottleneck shifting from error gradient computation to data-layout transformations, but the goodput is still much higher than Unfold+Parallel-GEMM as we see next.

Fig. 4f shows the relative speedup of our sparse kernel over Unfold+Parallel-GEMM. With sparsity $>= 0.75$, we consistently outperform. With sparsity of $>= 0.90\%$, we are significantly faster with 3x-32x speedup. The much larger improvement for blue and green kernels (first and last) is not just due to sparsity but also because Sparse-Kernel (BP) does not reduce AIT of the convolution by casting it into MM as described in Sec. 3.1.

4.3 Stencil-Kernel

We develop a new approach inspired by stencil computations [19] to improve single-core performance of small CNNs due to low AIT caused by the unfolding process, which we refer to as Stencil-Kernel. It computes CNNs through direct convolution without unfolding, exploits spatial reuse of inputs and improves AIT. In stencil computation [19], each element of an array is updated based on the neighbouring values specified by the stencil. For example, consider the following three-point stencil in one dimension:

$$A[x] = W_0 A[x] + W_1 A[x+1] + W_2 A[x+2]\,. \tag{16}$$

Each element of A is used to compute three output elements (e.g., A[x+2] is used to compute $A[x]$, $A[x+1]$ and $A[x+2]$). This *spatial reuse* property of stencils improves AIT by reusing an input value, while it is in fast memory, to compute multiple output values. Convolutions also exhibit spatial reuse since each input neuron contributes to multiple neighboring output neurons. Thus, stencil computation is more efficient for convolutions than unfolding which destroys spatial reuse by replicating the input neurons. Although there is a rich body of work exists on optimizing stencil computations [28, 29, 35, 46, 47], we present the first work to extend those optimization techniques for training CNNs on CPUs.

We illustrate how to use stencil computation for FP with the following example:

$$O[f,y,x] = \sum_{c,ky,k_x} I[c, y+k_y, x+k_x] \times W[f,c,k_y,k_x] \quad (17)$$

$$= \sum_c \Big(\sum_{k_y,k_x} I[c, y+k_y, x+k_x] \times W[f,c,k_y,k_x] \Big) \quad (18)$$

$$= \sum_c (S[f,c,y,x]) \quad (19)$$

where, O, I, W are output activations, input activations and weights, respectively. For a given y, x, c and f, the computation inside the parenthesis in Eqn. 18 is a two dimensional $F_x \times F_y$ point box stencil operation. $S[f,c,y,x]$ represents the result of this stencil operation. For a given f, c we can simplify and write the stencil operation as

$$S[y,x] = \sum_{k_y,k_x} I[y+k_y, x+k_x] \times W[k_y,k_x]) \,. \quad (20)$$

Our FP kernel generator uses this stencil operation as the building block for generating efficient vector code. It consists of two components: i) a basic block generator, ii) a schedule generator. The basic block generator generates register tiled vector instructions to improve the reuse of each input vector load and reduce the total number of load instructions. The schedule generator tiles the generated computation blocks to optimize for cache locality and TLB misses.

Basic Block Generator: The basic block generator vectorizes the stencil computation along x dimension and generates the instructions within a block as follows. For an output vector register tile of width r_x and height r_y, the block generator identifies all the input vectors that contribute to the tile. For each input vector, it generates instructions for loading it, and for computing its contributions to all the output vectors in the register tile. Fig. 7 shows the basic computation block for a convolution with $F_x = 1$ and $F_y = 2$, and a register tile size of $r_x = 1$ and $r_y = 2$. The block consists of three vector loads. The load of vector $ivec0$ only contributes to one output vector $ovec[0][0]$ in the register tile, while the load of $ivec1$ contributes to two vectors $ovec[0][0]$ and $ovec[0][1]$ in the output register tile. In other words, the load of $ivec1$ is reused twice. The shape and size of the register tile can change the overall reuse of each input vector load. In general, the size of r_x and r_y should be chosen such that $r_x r_y \leq$ number of physical vector registers and the number of load instructions is minimized. While this is a geometric optimization problem, our code generator finds optimal solution by iterating over all possible values for r_x and r_y meeting the first criteria as commodity machines typically have a relatively small number of vector registers.

```
/*load input vector 0 and compute 1 contribution*/
m256 ivec0 = mm256_loadu(input + (y + 0)*NX + x);
wvec[0][0] = mm256_set1(weight[0]);
m256 temp0 = mm256_mul(ivec0, wvec0[0][0]);
ovec[0][0] = mm256_add(ovec[0][0], tmpvec);

/*load input vector 1 and compute 2 contributions*/
m256 ivec1 = mm256_loadu(input + (y + 1)*NX + x);
wvec[0][1] = mm256_set1(weight[1]);
m256 temp1 = mm256_mul(ivec1, wvec[0][1]);
ovec[0][0] = mm256_add(ovec[0][0], temp1);
m256 temp2 = mm256_mul(ivec1, wvec[0][0] );
ovec[0][1] = mm256_add(ovec[0][1], temp1);

/*load input vector 2 and compute 1 contribution*/
m256 ivec2 = mm256_loadu(input + (y + 2)*NX + x);
m256 temp3 = mm256_mul(ivec2, wvec0[0][1]);
ovec[0][1] = mm256_add(ovec[0][1], temp3);
```

Figure 7: Basic code block for 1×2 stencil with a register tile size of $r_x = 1$ and $r_y = 2$

Strided Convolutions: The code generation technique above works well for convolutions with unit stride along x dimension; the convolutions with non-unit stride are more challenging because strided access can hinder effective vectorization. More specifically, for efficient vectorization, the inputs corresponding to an output vector should be contiguous in memory so that a single vector load instruction can load the inputs from memory to a vector register. However, any non-unit stride along x dimension will require loading inputs into a vector that is not contiguous in memory. Thus, our code generator performs a data-layout transformation to get the required input contiguous in memory for effective vectorization. For a given stride s_x, the layout of the input is transformed as

$$I[f,y,x] \rightarrow I[f,y,s,x'] \quad (21)$$

such that $s = x \ mod \ s_x$, $x' = x/s_x$ and $\frac{N_x}{s_x}s + x' = x$, where N_x is the size of the x dimension. This data-layout transformation, inspired by [28], converts unaligned vector loads into aligned vector loads.

Schedule Generator: Locality optimizations are used to reduce TLB and cache misses. Corresponding input and output are copied into contiguous memory to reduce the required number of TLB entries for accessing them, and then tiled so that input and output tiles fit in cache.

Evaluation: Fig. 4c shows scalability and absolute performance of Stencil-Kernel(FP) (including the data-layout transformation time). We see that Stencil-Kernel (FP) scales

Table 2: Convolution specifications for different benchmarks : $N_x(=N_y), N_f, N_c, F_x(=F_y), s_x(=s_y)$. The disparity in N_x values of Layer 0 is due to image padding/cropping.

Layer ID	ImageNet 22K	ImageNet 1K	CIFAR-10	MNIST
0	262,120,3,7,2	224,96,3,11,4	36,64,3,5,1	28,20,1,5,1
1	64,250,120,5,2	55,256,96,5,1	8,64,64,5,1	
2	15,400,250,3,1	27,384,256,3,1		
3	13,400,400,3,1	13,256,192,3,1		
4	11,600,400,3,1			

better than GEMM-in-Parallel (Figure 3a) as the impact of increasing core count on the performance per core is small.

Fig. 4d compares the performance of Stencil-Kernel(FP) and GEMM-in-Parallel. Stencil-Kernel(FP) outperforms GEMM-in-Parallel for small convolutions ($<$ 128 output features) because it improves their AIT; GEMM-in-Parallel performs better for larger convolutions, which already have large AIT.

4.4 Putting it all together

spg-CNN integrates the three techniques and automatically identifies the best set for each convolution layer of CNNs with different characteristics. It runs each layer with Parallel-GEMM, GEMM-in-Parallel, and Stencil-Kernel(FP) for FP and Parallel-GEMM, GEMM-in-Parallel and Sparse-Kernel(BP) for BP. Based on the measured performance, it chooses the fastest technique to deploy for each layer. For BP, it checks for a change in relative performance between these techniques after a pre-specified number of epochs as error gradient sparsity changes during the training. This ensures performance optimality at all training phases.

For our implementation and machine, GEMM-in-Parallel scales better than Parallel-GEMM for layers with < 1024 features, Sparse-Kernel(BP) is faster than GEMM-in-Parallel for layers with $> 75\%$ sparsity, and Stencil-Kernel(FP) is faster than GEMM-in-Parallel for layers with < 128 output features. These numbers are sensitive to the parameters of the implementation and the machine.

5. Experimental Evaluation

We implement and evaluate our framework on state-of-the-art training platforms and show its performance improvement on the convolutional layers of four well-known benchmarks, as well as end-to-end performance of training CIFAR.

5.1 Methodology

Benchmarks: We use four popular image recognition CNNs in our experiments: MNIST (LeCunn) [40], CIFAR-10 [50], ImageNet-1K (AlexNet) [38], and ImageNet-22K (Adam-ImageNet) [12]. The convolutional layer specifications for these benchmarks, based on the corresponding publications, are summarized in Table 2. The MNIST model classifies 28x28 black-and-white hand-written digits into 10 categories, CIFAR-10 classifies 32x32 RGB images into 10 categories, while ImageNet-1K and ImageNet-22K classify 256x256 RGB images into 1000 and 22000 categories.

Training Platforms: We use the ADAM [12] and CAFFE [33] CNN training platforms to obtain baseline results of the conventional approach, which we label as Parallel-GEMM (ADAM) and Parallel-GEMM (CAFFE). The parallel-GEMM implementations in ADAM and CAFFE use Intel MKL [30] and OpenBLAS [54] GEMM libraries respectively. Our results show that the limitation of the conventional approach is independent of the specific training platforms and GEMM library implementations.

Our Framework: We implement our framework on top of ADAM. To show overall performance as well as the incremental contributions of individual techniques, we present the results for i) GEMM-in-Parallel for both FP and BP ii) GEMM-in-Parallel for FP and Sparse-Kernel for BP, and ultimately, iii) Stencil-Kernel for FP and Sparse-Kernel for BP.

Hardware: All of the experiments were run on a Xeon Intel(R) Xeon(R) CPU E5-2650 with 16 physical cores (32 logical cores with hyper-threading enabled).

5.2 Improvements of Real World Convolutions

Fig. 8 shows performance improvement using our framework over Parallel-GEMM for convolution layers in the four real-world benchmarks. We achieve 2x - 16x speedup on FP. More specifically, for ImageNet 22K and 1K, the speed up is a result of using GEMM-in-Parallel. These convolutions, with the number of output features ranging from 96 to 384, do not have enough AIT for Parallel-GEMM to be effective, but enough for GEMM-in-Parallel to perform well. For CIFAR and MNIST, we achieve even higher speedup than ImageNet22K and 1K, by using Stencil-Kernel. Their number of features ranges between 20 and 64, resulting in low AIT. While using GEMM-in-Parallel improves AIT, it is still not highly effective. Stencil-Kernel increases AIT further and achieves higher performance. Take layer 1 of CIFAR as an example: speedup of GEMM-in-Parallel over Parallel-GEMM is 11.5x (shown as blue bar in Fig. 8), while Stencil-Kernel boosts the speedup further to 16x (green bar). For smaller CNNs like MNIST, both Parallel-GEMM and GEMM-in-Parallel perform poorly while Stencil-Kernel achieves 9x speedup over them.

Last but not the least, Fig. 8 also shows performance improvement on BP using our Sparse-Kernel. These speedup values are based on 85% sparsity in errors. We pick this sparsity level conservatively based on Fig. 3b. We achieve speedup of $2-14$x over the baselines.

5.3 End-to-End evaluation on CIFAR-10 training

Our framework accelerates the end-to-end training of CIFAR-10 by 12.3x and 8.3x compared to Parallel-GEMM(ADAM) and Parallel-GEMM(CAFFE) respectively. Figure 9 reports the peformance (throughput) of the different techniques in terms of images trained per second. The x-axis shows the number of cores used for training.

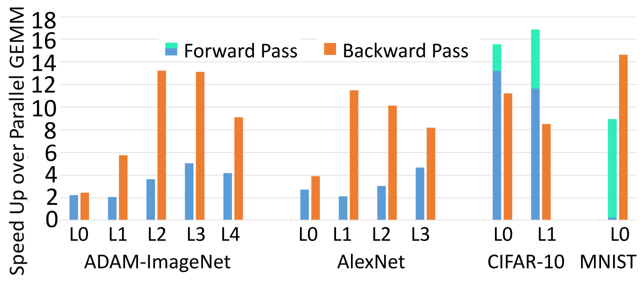


Figure 8: Performance improvement using our framework over Parallel-GEMM for convolution layers in real world benchmarks. For FP, the speedup of GEMM-in-Parallel over Parallel-GEMM is shown in blue, and if there is any, the additional speedup from Stencil-Kernel is shown in green. For BP, the speedup of Sparse-Kernel is shown in orange.

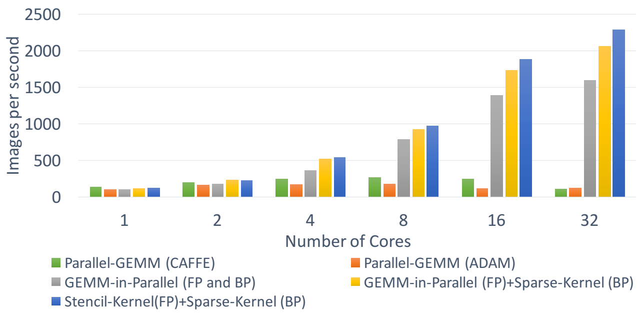


Figure 9: End-to-end performance comparison in CIFAR.

For one and two cores, Parallel-GEMM (CAFFE) is the fastest, but for more than two cores, both Parallel-GEMM (CAFFE) and Parallel-GEMM (ADAM) stop scaling — due to decrease in AIT per core as discussed in Sec. 4.1. For more than 2 cores, GEMM-in-Parallel scales better than both Parallel-GEMM (CAFFE) and Parallel-GEMM (ADAM) — GEMM-in-Parallel does not reduce AIT as we increase the number of cores.

Fig. 9 also shows the performance improvement using Sparse-Kernel on BP and Stencil-Kernel on FP. At 32 cores, Sparse-Kernel on BP increases the number of images per second from $1,600$ to $2,061$, improving throughput of about 28%. Adding Stencil-Kernel to FP increases the throughput to $2,283$, a further 10% improvement. Notice these are net improvements on the end-to-end performance of the system, while the improvement on convolution layers is even higher, up to 30%, as seen in Fig. 8.

Summary: Parallel-GEMM (CAFFE) and Parallel-GEMM (ADAMS) peak performance are 273 and 185 images per second, respectively. Using Stencil-Kernel (FP) and Sparse-Kernel (BP), we increase the training throughput to $2,283$, a net speedup of 8.36x. To put this in perspective, it takes Parallel-GEMM (CAFFE) 36 mins to train our model, while the optimized version takes only 4.3 minutes.

6. Related Work

This section reviews the related work on optimizing CNNs from various perspectives.

Parallelism. There are two forms of parallelism for CNN training on multicore CPUs, parallel-GEMM and GEMM-in-parallel. Many deep learning frameworks such as CAFFE, TensorFlow, Theano and Torch 7 use the first approach. Our study presents the first performance characterization of parallel-GEMM vs. GEMM-in-parallel for CNNs, and we show that while both approaches are similar in Region 0, GEMM-in-parallel is faster in Region 2, especially with increased number of cores. Alternatively, Caffe con Troll[6] improves Parallel-GEMM performance in Region 2 by executing a batch of image partitions (rather than one partition) per core.

Sparsity. [51, 52] were the first to propose reducing computation in multilayer perceptrons by simply avoiding calculations with zero values, but they do not discuss how to implement this efficiently on modern architectures. The poster [25] describes a sparse dense MM algorithm using compressed sparse row to store the sparse error/activation matrix. Their approach and evaluation is limited to large sparse MM that are not applicable to real CNNs. [42] also presents a sparse dense MM algorithm, which exploits sparsity in the weights. Their algorithm is based on knowing the position of non-zero elements in weights in advance to generate the sparse MM code, therefore, their approach is only applicable for CNN inference but not training. In contrast, our approach is designed for training and we exploit sparsity in activation errors. Additionally, instead of using sparse MM we compose a sparse convolution as a series of small and dense MMs, achieving high goodput on sparse dense inputs.

Direct convolution. Although CNNs can be computed on CPUs via direct convolution, most CNN libraries/frameworks in practice use GEMM instead because (1) direct convolution is much harder to optimize due to its more complex computation structure and (2) highly optimized GEMM implementations are widely available, achieving significant fraction of peak hardware performance for large CNNs. However, as we pointed out in Section 3, for small CNNs, GEMM based approach has a lower AIT compared to direction convolution.

Our stencil kernel is designed for small CNNs, performing direct convolution. Beyond that, we optimize vector register reuse and L1 cache locality by exploiting spatial reuse in convolutions. Other CNN frameworks such as cuDNN [11], NeuFlow [24], implement direct convolution on GPUs and custom hardware, but not on CPUs. On CPUs, Intel Deep Learning Framework (IDLF) [4] does direct convolution without the stencil-based optimzations we perform. Additionally, it only supports AVX2 ISA based CPUs, making it unavailable for abundantly available legacy CPU hardware.

Distributed training platforms. There are complementary efforts to train large CNNs on a cluster of distributed machines with multicore CPUs, for example, Google's DistBelief [20] and Microsoft's Adam [12]). They exploit massively parallel architectures through data parallelism. Specifically, to expedite CNN training on very large data sets, many worker machines train in parallel on different subsets of the training data. Each worker periodically synchronizes its model parameters with other workers. The time to train a model is therefore a function of the throughput of the worker machines (inputs processed per second) and the latency of synchronizing model parameters. Our work, focusing on CNN optimization on multicore CPUs, could improve the throughput of each worker machine, and therefore help to accelerate the training of large CNNs that are compute bound (e.g., image models).

Hardware Accelerators. There is rich complementary work of optimizing CNNs on hardware accelerators. GPUs [27], FPGAs [23, 45, 55] and ASICs [9, 10] have being used to accelerate CNN inference, which is a subset (i.e., forward propagation phase only) of CNN training. Cuda-Convnet [36] provides fast implementation of convolution on NVIDIA GPUs, and several studies [14, 37] exploit multi-GPU clusters to speed up CNN training. Our observation of exploiting sparsity could be applicable on some of these accelerators to improve goodput and speed up CNN training, which might be an interesting area of future work. While optimizing CNNs on GPUs, FPGAs, and ASICs is important, the abundance and easy accessibility of CPU based systems makes CNN optimizations for CPUs vital.

Other techniques. There are other complementary efforts to optimize CNN, e.g., by exploiting redundancies [21, 22, 31, 42], using FFT-based computation [44], and applying more efficient MM computation [18].

7. Conclusion

This paper presents the first characterization of the optimization opportunities for training CNNs on CPUs. Given this characterization, we have designed and implemented an optimization framework *spg*-CNN that beats the state-of-the-art approaches to training CNNs by an order of magnitude. In an environment where competitive CNN models may take weeks to train, our results enable CNN model engineers to iterate an order-of-magnitude more quickly through the multiple model designs that are often required to identify a good model for one's domain.

References

[1] `http://chainer.org`.

[2] `http://www.cntk.ai`.

[3] `https://developers.google.com/protocol-buffers/`.

[4] `https://01.org/intel-deep-learning-framework`.

[5] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous systems, 2015. *Software available from tensorflow. org*.

[6] F. Abuzaid, S. Hadjis, C. Zhang, and C. Ré. Caffe con troll: Shallow ideas to speed up deep learning. *arXiv preprint arXiv:1504.04343*, 2015.

[7] F. Bastien, P. Lamblin, R. Pascanu, J. Bergstra, I. J. Goodfellow, A. Bergeron, N. Bouchard, and Y. Bengio. Theano: new features and speed improvements, 2012.

[8] K. Chellapilla, S. Puri, and P. Simard. High performance convolutional neural networks for document processing. In *Suvisoft Tenth International Workshop on Frontiers in Handwriting Recognition*, 2006.

[9] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014.

[10] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, et al. Dadiannao: A machine-learning supercomputer. In *IEEE/ACM International Symposium on Microarchitecture*, 2014.

[11] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

[12] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *11th USENIX Symposium on Operating Systems Design and Implementation*, 2014.

[13] D. Ciresan, U. Meier, and J. Schmidhuber. Multi-column deep neural networks for image classification. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2012.

[14] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew. Deep learning with cots hpc systems. In *Proceedings of the 30th International Conference on Machine Learning*, 2013.

[15] R. Collobert and J. Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *ACM Proceedings of the 25th international conference on Machine learning*, 2008.

[16] R. Collobert, K. Kavukcuoglu, and C. Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, Neural Information Processing Systems Workshop*, number EPFL-CONF-192376, 2011.

[17] 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.

[18] J. Cong and B. Xiao. Minimizing computation in convolutional neural networks. In *Springer International Conference on Artificial Neural Networks*. 2014.

[19] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, 2008.

[20] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le, et al. Large scale

distributed deep networks. In *Advances in Neural Information Processing Systems*, 2012.

[21] M. Denil, B. Shakibi, L. Dinh, N. de Freitas, et al. Predicting parameters in deep learning. In *Advances in Neural Information Processing Systems*, 2013.

[22] E. L. Denton, W. Zaremba, J. Bruna, Y. LeCun, and R. Fergus. Exploiting linear structure within convolutional networks for efficient evaluation. In *Advances in Neural Information Processing Systems*, 2014.

[23] C. Farabet, C. Poulet, J. Y. Han, and Y. LeCun. Cnp: An fpga-based processor for convolutional networks. In *International Conference on Field Programmable Logic and Applications*, 2009.

[24] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun. Neuflow: A runtime reconfigurable dataflow processor for vision. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, 2011.

[25] Y. Gao, Y. Liu, R. Zhao, and S. Chiu. An efficient sparse matrix multiplication for deep neural network-based applications. 2014.

[26] K. Goto and R. A. Geijn. Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software*, 2008.

[27] J. Hauswald, Y. Kang, M. A. Laurenzano, Q. Chen, C. Li, T. Mudge, R. G. Dreslinski, J. Mars, and L. Tang. Djinn and tonic: Dnn as a service and its implications for future warehouse scale computers. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2015.

[28] T. Henretty, K. Stock, L.-N. Pouchet, F. Franchetti, J. Ramanujam, and P. Sadayappan. Data layout transformation for stencil computations on short-vector simd architectures. In *Springer International Conference on Compiler Construction*, 2011.

[29] J. Holewinski, L.-N. Pouchet, and P. Sadayappan. High-performance code generation for stencil computations on gpu architectures. In *Proceedings of the 26th ACM international conference on Supercomputing*, 2012.

[30] M. Intel. Intel math kernel library, 2007.

[31] M. Jaderberg, A. Vedaldi, and A. Zisserman. Speeding up convolutional neural networks with low rank expansions. *arXiv preprint arXiv:1405.3866*, 2014.

[32] S. Ji, W. Xu, M. Yang, and K. Yu. 3d convolutional neural networks for human action recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2013.

[33] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the ACM International Conference on Multimedia*, pages 675–678, 2014.

[34] A. Karpathy, G. Toderici, S. Shetty, T. Leung, R. Sukthankar, and L. Fei-Fei. Large-scale video classification with convolutional neural networks. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2014.

[35] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan. Effective automatic parallelization of stencil computations. In *ACM Sigplan Notices*, 2007.

[36] A. Krizhevskey. Cuda-convnet, 2014.

[37] A. Krizhevsky. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997*, 2014.

[38] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, 2012.

[39] Q. V. Le, W. Y. Zou, S. Y. Yeung, and A. Y. Ng. Learning hierarchical invariant spatio-temporal features for action recognition with independent subspace analysis. 2011.

[40] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 1998.

[41] H. Lee, R. Grosse, R. Ranganath, and A. Y. Ng. Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations. In *ACM Proceedings of the 26th Annual International Conference on Machine Learning*, 2009.

[42] B. Liu, M. Wang, H. Foroosh, M. Tappen, and M. Pensky. Sparse convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015.

[43] W. Liu and B. Vinter. A framework for general sparse matrix-matrix multiplication on gpus and heterogeneous processors. *arXiv preprint arXiv:1504.05022*, 2015.

[44] M. Mathieu, M. Henaff, and Y. LeCun. Fast training of convolutional networks through ffts. *arXiv preprint arXiv:1312.5851*, 2013.

[45] K. Ovtcharov, O. Ruwase, J.-Y. Kim, J. Fowers, K. Strauss, and E. S. Chung. Accelerating deep convolutional neural networks using specialized hardware, 2015.

[46] L. Peng, R. Seymour, K.-i. Nomura, R. K. Kalia, A. Nakano, P. Vashishta, A. Loddoch, M. Netzband, W. R. Volz, and C. C. Wong. High-order stencil computations on multicore clusters. In *IEEE International Symposium on Parallel & Distributed Processing*, 2009.

[47] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices*, 2013.

[48] P. Y. Simard, D. Steinkraus, and J. C. Platt. Best practices for convolutional neural networks applied to visual document analysis. In *IEEE International Conference on Document Analysis and Recognition*, 2003.

[49] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[50] 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.

[51] F. Wang and Zhang. An adaptive and fully sparse training approach for multilayer perceptrons, 1996.

[52] F. Wang and Q. Zhang. A sparse matrix approach to neural network training. In *Proceedings of the IEEE International Conference on Neural Network*, 1995.

[53] R. C. Whaley. Atlas (automatically tuned linear algebra software). In *Springer Encyclopedia of Parallel Computing*. 2011.

[54] Z. Xianyi, W. Qian, and Z. Chothia. Openblas. *URL: http://xianyi. github. io/OpenBLAS*, 2012.

[55] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2015.