SparseTrain: Leveraging Dynamic Sparsity in Software for Training DNNs on General-Purpose SIMD Processors

Zhangxiaowen Gong
gong15@illinois.edu
University of Illinois at Urbana-Champaign
Champaign, IL, USA

Houxiang Ji
hj14@illinois.edu
University of Illinois at Urbana-Champaign
Champaign, IL, USA

Christopher W. Fletcher
cwfletch@illinois.edu
University of Illinois at Urbana-Champaign
Champaign, IL, USA

Christopher J. Hughes
christopher.j.hughes@intel.com
Intel Labs
Santa Clara, CA, USA

Josep Torrellas
torrella@illinois.edu
University of Illinois at Urbana-Champaign
Champaign, IL, USA

ABSTRACT

Our community has improved the efficiency of deep learning applications by exploiting sparsity in inputs. Most of that work, though, is for inference, where weight sparsity is known statically, and/or for specialized hardware. In this paper, we propose *SparseTrain*, a *software-only* scheme to leverage dynamic sparsity during training on general-purpose SIMD processors. *SparseTrain* exploits zeros introduced by the ReLU activation function to both feature maps and their gradients. Exploiting such sparsity is challenging because the sparsity degree is moderate and the locations of zeros change over time.

SparseTrain identifies zeros in a dense data representation and performs vectorized computation. Variations of the scheme are applicable to all major components of training: forward propagation, backward propagation by inputs, and backward propagation by weights. Our experiments on a 6-core Intel Skylake-X server show that *SparseTrain* is very effective. In end-to-end training of VGG16, ResNet-34, and ResNet-50 with ImageNet, *SparseTrain* outperforms a highly-optimized direct convolution on the non-initial convolutional layers by 2.19x, 1.37x, and 1.31x, respectively. *SparseTrain* also benefits inference. It accelerates the non-initial convolutional layers of the aforementioned models by 1.88x, 1.64x, and 1.44x, respectively.

CCS CONCEPTS

• **Computing methodologies** → **Neural networks**; *Shared memory algorithms*; *Vector / streaming algorithms.*

KEYWORDS

Deep neural networks, training, convolution, sparsity, CPU

PACT '20, October 3–7, 2020, Virtual Event, GA, USA

ACM ISBN 978-1-4503-8075-1/20/10…$15.00
https://doi.org/10.1145/3410463.3414655

ACM Reference Format:
Zhangxiaowen Gong, Houxiang Ji, Christopher W. Fletcher, Christopher J. Hughes, and Josep Torrellas. 2020. SparseTrain: Leveraging Dynamic Sparsity in Software for Training DNNs on General-Purpose SIMD Processors. In *Proceedings of the 2020 International Conference on Parallel Architectures and Compilation Techniques (PACT '20), October 3–7, 2020, Virtual Event, GA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3410463.3414655

1 INTRODUCTION

Deep Neural Networks (DNNs) have become ubiquitous, achieving state-of-the-art results across a range of tasks from image recognition [26] to speech recognition [5], scene generation [37], and game playing [42]. While GPUs are amongst the fastest hardware solutions today for DNN training, CPUs are also popular platforms because they have already been widely deployed in datacenter, client, and edge devices, therefore lowering the Total Cost of Ownership (TCO) for the DNN market [2, 39, 47, 52]. The large memory capacity on the CPU platforms (e.g., up to 4.5TB per socket with the third generation Intel Xeon Scalable processors) also makes training with large datasets and/or models easier [52]. Consequently, industry often uses a significant number of datacenter CPUs available during off-peak periods to do distributed training. For example, Facebook trains their "Sigma" product entirely on CPUs and their "Facer" product partially on CPUs [16]. Other examples of training on CPUs include Intel's assembly and test factory, deepsense.ai's reinforcement learning (RL), Kyoto University's drug design, Clemson University's natural language processing (NLP), GE Healthcare's medical imaging, and many more [39]. Previous works [8, 11] have already demonstrated good strong scaling of distributed training on clusters of CPU nodes; hence, a good way to further reduce training time is to accelerate the work assigned to each CPU node [11]. Overall, accelerating DNN training on general-purpose processors is an important yet sometimes undervalued task.

An effective approach to accelerating DNNs is to remove useless computations on zero values in the data, known as *sparsity*. Indeed, prior efforts spanning hardware to software and algorithms have exploited sparsity to eliminate computation or data transfers at different points in DNN computations. Most of these efforts, though, require hardware changes [3, 7, 13, 34, 38, 40, 57] and/or apply only to *inference* [3, 7, 13, 15, 34, 35, 50, 53, 57]. This is not ideal,

since most of real-world DNN computations are performed on conventional CPUs and GPUs [4, 16, 33, 51], and significant time goes into *training*.

This paper addresses these shortcomings through a *software only* effort to speed up DNN training leveraging sparsity, on unmodified general-purpose CPUs. This is challenging for multiple reasons. First, works targeting sparse inference typically rely on sparse representations (e.g., Compressed Sparse Row, or CSR), leveraging *static* sparsity patterns (i.e., the locations of the non-zeros) [13, 34, 35, 50, 53, 57]. This is reasonable in inference because the DNN weights do not change. In training, though, the sparsity pattern in both inputs and weights changes over time, since the weights are updated with each batch of inputs. Second, operating on sparse data incurs overhead: modern machines are highly optimized for dense computations, and suffer from the extra indirections and branches that appear when processing sparse data. Prior work either relies on custom hardware to minimize these overheads [3, 7, 13, 34, 40, 57], or sophisticated pre-processing to "shape" the sparsity pattern to better match existing hardware [35, 50, 53]—which only applies to static sparsity.

Our software scheme to exploit dynamic sparsity on general-purpose SIMD processors is called *SparseTrain*. *SparseTrain* leverages the rectified linear unit (ReLU [30]), a ubiquitous operator used by convolutional neural networks (CNNs) [17, 19, 20, 26, 43, 46], multilayer perceptrons (MLPs) [23], and recurrent neural networks (RNNs) [5]. After each ReLU-activated DNN layer, all neurons (outputs) in the layer are clamped to zero if negative. Whether a neuron is negative depends on the inputs and weights, both of which change during training. Thus, ReLU introduces *dynamic sparsity*.

However, ReLU only induces 40%-90% sparsity [38], which is moderate compared to many sparse scientific computations. Further, the sparsity pattern has no discernible structure. Hence, *SparseTrain* operates on *data in a dense format*. It exploits sparsity by detecting zero input values at runtime, and, when appropriate, branching over useless computations such as multiply-by-zero.

The amount of computation that can be skipped due to a zero neuron depends on the number of reuse of the neuron. High reuse helps amortize the overheads incurred while detecting and exploiting sparsity. Among different types of DNNs, CNNs have the highest reuse of their neurons. Therefore, while the approach is generally applicable to any DNN employing ReLU, we focus on CNNs in this paper. Furthermore, *SparseTrain* introduces optimizations to minimize overhead while maximizing data locality, available parallelism, and the amount of work skipped per zero input.

Our experiments on a 6-core Intel Skylake-X server show that *SparseTrain* is very effective. In end-to-end training of VGG16, ResNet-34, and ResNet-50 with ImageNet, *SparseTrain* outperforms a highly-optimized direct convolution on the non-initial convolutional layers by 2.19x, 1.37x, and 1.31x, respectively. *SparseTrain* also benefits inference. It accelerates the non-initial convolutional layers of the aforementioned models by 1.88x, 1.64x, and 1.44x respectively.

We make the following contributions:

- The development of *SparseTrain*, the first approach to exploit dynamic sparsity during DNN training on general-purpose CPUs in software.
- A novel sparse algorithm that does not rely on a sparse representation and is effective for moderate sparsity.
- An evaluation showing that *SparseTrain* outperforms a highly-optimized direct convolution on multiple DNNs.

2 BACKGROUND

2.1 Training Convolutional Neural Networks

A CNN is a type of DNN that is effective for analyzing images. The leading competitors in recent years' ImageNet Large Scale Visual Recognition Competition (ILSVRC) are mostly variants of CNNs, such as AlexNet [26], VGG [43], GoogLeNet [46], and ResNet [17]. Within a CNN, the convolutional (i.e., *conv*) layers are the most time consuming components; thus, reducing the amount of computation in them can greatly boost performance. In the following discussion, we use the symbols listed in Table 1.

Table 1: List of the symbols and their dimensions & iterators.

	Description	Itr.		Description	Dimension	Iterator
N	minibatch size	i	D	input tensor	$NCWH$	i, c, x, y
C	input channels	c	Y	output tensor	$NKW'H'$	i, k, x', y'
K	output channels	k	G	weight tensor	$KCRS$	k, c, u, v
W	input width	x	L	loss function		
H	input height	y	V	vector length		
R	filter width	u	T	# of skippable ops		
S	filter height	v	M	minibatch tile size		
O	horizontal stride		Q	output channel tile		
P	vertical stride			size		

The convolution on a minibatch of N images with C channels and size $H \times W$ correlates a set of K filters with C channels and size $S \times R$ on the images, producing a minibatch of N images with K channels and size $H/P \times W/O$, where P and O are the strides of the two dimensions, respectively. We denote filter elements as $G_{k,c,u,v}$ and image elements as $D_{i,c,x,y}$. The forward convolution for output $Y_{i,k,x',y'}$ is:

$$Y_{i,k,x',y'} = \sum_{c=0}^{C-1} \sum_{u=0}^{R-1} \sum_{v=0}^{S-1} D_{i,c,x' \times O+u,y' \times P+v} \times G_{k,c,u,v} \quad (1)$$

In the backward propagation of a convolutional layer, the gradient of the loss function L with respect to the weights G is calculated by applying the chain rule:

$$\frac{\partial L}{\partial G} = \frac{\partial L}{\partial Y} \frac{\partial Y}{\partial G} \quad (2)$$

We need $\partial L/\partial Y$ from the next layer, and compute $\partial L/\partial D$ for the previous layer if needed. $\partial L/\partial D$ is a convolution of $\partial L/\partial Y$ with the layer's filters transposed. The gradient with respect to the weights is a convolution of D with $\partial L/\partial Y$, producing $S \times R$ outputs for each input/output channel combination.

Training a conv layer has three major components: the forward propagation (FWD), the backward propagation by input (BWI), and the backward propagation by weights (BWW). Table 2 lists the parameters of the layers that we evaluate.

Table 2: Evaluated layer configurations from VGG and ResNet v1.5.

Name	C	K	H	W	R	S	O	P	Name	C	K	H	W	R	S	O	P	Name	C	K	H	W	R	S	O	P
vgg1_2	64	64	224	224	3	3	1	1	vgg2_1	64	128	112	112	3	3	1	1	vgg2_2	128	128	112	112	3	3	1	1
vgg3_1	128	256	56	56	3	3	1	1	vgg3_2	256	256	56	56	3	3	1	1	vgg4_1	256	512	28	28	3	3	1	1
vgg4_2	512	512	28	28	3	3	1	1	vgg5_1	512	512	14	14	3	3	1	1	resnet2_1a	64	64	56	56	1	1	1	1
resnet2_1b	256	64	56	56	1	1	1	1	resnet2_2	64	64	56	56	3	3	1	1	resnet2_3	64	256	56	56	1	1	1	1
resnet3_1a	256	128	56	56	1	1	1	1	resnet3_1b	512	128	28	28	1	1	1	1	resnet3_2	128	128	28	28	3	3	1	1
resnet3_2/r	128	128	56	56	3	3	2	2	resnet3_3	128	512	28	28	1	1	1	1	resnet4_1a	512	256	28	28	1	1	1	1
resnet4_1b	1024	256	14	14	1	1	1	1	resnet4_2	256	256	14	14	3	3	1	1	resnet4_2/r	256	256	28	28	3	3	2	2
resnet4_3	256	1024	14	14	1	1	1	1	resnet5_1a	1024	512	14	14	1	1	1	1	resnet5_1b	2048	512	7	7	1	1	1	1
resnet5_2	512	512	7	7	3	3	1	1	resnet5_2/r	512	512	14	14	3	3	2	2	resnet5_3	512	2048	7	7	1	1	1	1

2.2 ReLU and Dynamic Sparsity

Each output of a DNN layer is usually passed through an activation function to introduce non-linearity. One popular activation is ReLU:

$$f(x) = max(0, x) \tag{3}$$

and its derivative is[1]:

$$f'(x) = \begin{cases} 1, \text{ if } x > 0 \\ 0, \text{ otherwise} \end{cases} \tag{4}$$

By definition, ReLU and its derivative produce 50% sparsity when the distribution of x is centered at 0. When ReLU-activated conv layers are cascaded, this is reflected in D in the forward propagation and $\partial L/\partial Y$ in the backward propagation, and it affects all three training components.

Since ReLU-induced sparsity varies with input, we call it *dynamic* sparsity to differentiate it from the *static* sparsity of weight-pruning. Other sources of dynamic sparsity include dropout [44] and max pooling during backward propagation. Dynamic sparsity is the only type that exists during the majority of the training time.[2]

Exploiting dynamic sparsity is challenging because the level of sparsity is too low for a typical *irregular* sparse computation to outperform highly optimized *regular* dense computation. In addition, at modest sparsity, the metadata overheads of sparse representations such as CSR may exceed any savings.

2.3 Baseline Platform

We consider a shared-memory server comprising general-purpose processors with multiple cores and SIMD support. While we tune and evaluate on a specific platform described in Section 4, our approach is applicable to most modern shared-memory nodes with processors supporting SIMD. Further, our approach is fully compatible with multi-node implementations; it will simply accelerate the work done on each node.

To provide context for our design decisions, we briefly describe our baseline platform. We study a system with Intel Skylake cores. In each cycle, each core can execute two AVX-512 arithmetic instructions (e.g., vector fused multiply-add, or VFMA), read two cache lines (64B) and write one cache line from/to the L1 data cache, and retire four instructions. Each core has 32 vector registers, a 32KB L1 data cache, a 1MB L2 cache and a 1.375MB non-inclusive shared L3 cache.

[1]The derivative at $x = 0$ is undefined but usually set to 0.

[2]Static sparsity is also present when re-training a weight-pruned network, but we focus on regular dense training.

We implement our work as new convolution kernels in *MKL-DNN* [21], a highly tuned DNN library. We specialize the kernels according to the size of the convolution and the hardware parameters via just-in-time (JIT) compilation. Prior works also demonstrated that JIT-ing achieves higher performance than statically-tuned BLAS-calls for convolution [11, 18]. Because for a given convolutional layer, we only JIT the kernels once during the whole training process, the kernel generation overhead is virtually non-existent. Being low-level software, our implementation can be incorporated to DNN frameworks like *TensorFlow* [1] or *PyTorch* [36].

3 EXPLOITING DYNAMIC SPARSITY

We propose *SparseTrain*, which leverages dynamic sparsity to speed-up DNN training on shared-memory SIMD multiprocessors. The idea is to skip computations that are rendered ineffectual by ReLU. In particular, *SparseTrain* skips a multiply-accumulate operation (MAC) if one of its multiplicands is zero. *SparseTrain* uses a dense data representation for three reasons. First, the sparsity from ReLU is usually too low for any sparse representation to benefit. Second, we avoid the overhead of converting between dense and sparse representations. Finally, a dense format allows regular memory access patterns and more efficient vectorization.

In the following, we start by describing a naïve initial design, and then progressively improve it.

3.1 Naïve Forward Propagation (FWD)

We base our scheme on direct convolution. Algorithm 1 describes a naïve vectorized approach that skips computation in FWD upon detecting a zero input. Line 1 and Line 4 represent collapsed loop nests. For simplicity, the algorithm assumes unit stride, but can be easily expanded for strided convolution. In the rest of the paper, we assume unit stride unless otherwise specified. The sparse algorithm for BWI is similar to FWD, and we will talk about BWW separately.

The main idea is as follows. Since an input element is reused $R \times S \times K$ times, by making the input stationary in the computation loop nest, we may skip at most $R \times S \times K$ operations when we detect a zero. We vectorize the computation along the output channel dimension (K). The statement in Line 5 represents a VFMA operation of length V. When we detect a zero in Line 2, we skip all of the following $R \times S \times K/V$ ineffectual VFMAs. We denote the number of skippable VFMAs per check as T. As shown in Table 2, K is often on the order of hundreds. This, together with the reuse of $R \times S$ means that, potentially, T is large.

Algorithm 1: Naïve Vectorized Sparse FWD.

```
   input  : input D, filters G
   output : output Y
1  for i = 0, c = 0, y = 0, x = 0 to N − 1, C − 1, H − 1, W − 1 do
2    if D_{i,c,x,y} ≠ 0 then
3      for k = 0 to K − V step V do
4        for u = 0, v = 0 to R − 1, S − 1 do
5          Y_{i,[k:k+V−1],x−u,y−v} =
             Y_{i,[k:k+V−1],x−u,y−v} + D_{i,c,x,y} × G_{[k:k+V−1],c,u,v};
```

The naïve algorithm has several downsides. First, it naturally has input parallelism: it compares each D element to zero and then updates multiple Y elements. Input parallelization requires atomic updates of Y, which drastically reduces performance. Output parallelization is generally faster. The simplest such approach is to let each core work on different images in the minibatch. However, common practice on training on CPU clusters is to assign, to each multicore, only a small minibatch. As a result, it is likely that different cores will get a different number of images, resulting in load imbalance.

The second downside is that a CPU has a limited amount of ISA vector registers; this is 32 in the CPU we target. If $T = R \times S \times K/V$ is greater than the number of registers, we must spill registers during computation, inducing overhead. Therefore, we want to confine T within the register budget.

Finally, D has an unpredictable sparsity pattern, triggering frequent branch mispredictions in the zero-checking. Limiting T to the register budget (~32) reduces our chance to amortize the misprediction penalty.

3.2 Optimized Forward Propagation (FWD)

To improve the naïve FWD algorithm, we now introduce five optimizations. Algorithm 2 includes the high-level ideas.

Algorithm 2: Parallel Vectorized Sparse FWD.

```
   input  : input D, filters G
   output : output Y
1  for i = 0 to N − M step M in parallel do
2    for y = 0 to H − 1 in parallel do
3      for v = 0 to S − 1 do
4        for k = 0 to K − Q step Q in parallel do
5          for c = 0 to C − V step V do
6            for i' = i to i + M − 1 in parallel do
7              for x = 0 to W − 1 do
8                m_{[0:V−1]} = [d ≠ 0 for d in D_{i,[c:c+V−1],x,y+v}];
9                for c' = 0 to V − 1 do
10                 if m_{c'} is true then
11                   for k' = k to k + Q − V step V do
12                     for u = 0 to R − 1 do
13                       Y_{i',[k':k'+V−1],x−u,y} = Y_{i',[k':k'+V−1],x−u,y} +
                           D_{i',c+c',x,y+v} × G_{[k':k'+V−1],c+c',u,v};
```

3.2.1 Vectorized Zero-Checking. The naïve algorithm compares D elements to zero one at a time. To improve it, we vectorize this check along the input channel dimension (C). Specifically, Line 8 in Algorithm 2 does a vector comparison to zero to generate a vector boolean mask $m_{[0:V-1]}$; each mask bit is set if the corresponding input element is not zero. We then use the mask to determine whether to skip computation.

3.2.2 Increasing Output Parallelism. In a convolution, a D element affects a set of spatially-grouped Y elements. Similarly, a Y element is calculated from a limited set of spatially-grouped D elements. This allows us to increase output parallelism by reducing T.

We parallelize at an output row granularity. When a core works on an output row, it processes the D elements from S corresponding input rows, one row at a time. This approach lowers T from $R \times S \times K/V$ to $R \times K/V$. Moreover, if $R \times K/V$ is still larger than the number of ISA registers, we further reduce T to avoid register spilling. We accomplish this by tiling the output channel dimension (K) and decrease T to $R \times Q/V$, where Q is a factor of K and a multiple of V. We will discuss how we choose Q in the next section. We can process the same output row at different output channel tiles in parallel. With $T = R \times Q/V$, the number of parallel tasks rises from N in the naïve algorithm to $N \times H \times K/Q$.

Since an input row corresponds to S output rows, multiple cores may read a given input row. In a shared memory system, such reuse may be captured in a shared cache.

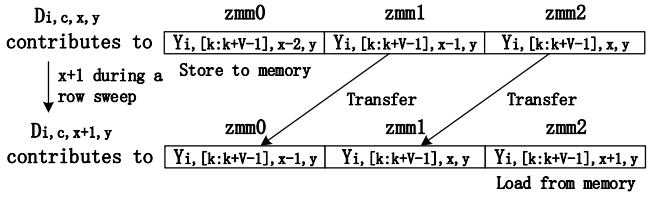
3.2.3 Efficient Vector Register Usage. A VFMA has three operands: one accumulator vector and two multiplicand vectors. In the target ISA, one multiplicand vector can be a memory operand. In modern Intel and AMD microarchitectures such as Skylake and Zen, the L1 read bandwidth matches the VFMA throughput (2 per cycle per core) [10]; thus, utilizing the memory operand does not slow down the computation.

When we translate Line 13 of Algorithm 2 to a VFMA instruction, we use the multiplicand vector $G_{[k':k'+V-1],c+c',u,v}$ as a memory operand. We broadcast $D_{i',c+c',x,y+v}$ to all lanes of a vector register and use the register as the other multiplicand vector. Note that all $T = R \times Q/V$ VFMAs in the loop from Lines 11-13 share this broadcasted D element. Finally, each VFMA needs a dedicated vector register to hold the accumulator vector $Y_{i',[k':k'+V-1],x-u,y}$. Therefore, we need $T + 1$ vector registers for a given T.

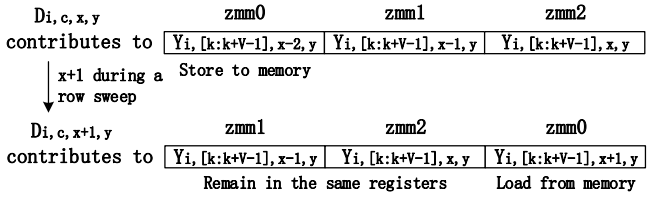
The target ISA has 32 `zmm` vector registers. Algorithm 2 keeps a vector of zeros for the vector compare instruction in Line 8. Hence, there are 31 vector registers available. Because we need $T + 1$ vector registers, we limit T to 30 in order not to spill the registers.

Besides avoiding register spilling, we further reduce memory operations. As shown in Lines 7-13, we scan through an input row and update the affected Y elements accordingly. We call such a scan a *Row Sweep*. Figure 1 illustrates examples of how we optimize both memory access and register usage during a row sweep.

Due to a convolution's spatial nature, adjacent D elements may contribute to overlapping Y elements, depending on the filter width R and the horizontal stride O. Consider the example in Figure 1a. When $R = 3$ and $O = 1$, $D_{i,c,x,y}$ contributes to $Y_{i,[k:k+V-1],[x-2:x],y}$. The next element $D_{i,c,x+1,y}$ contributes to $Y_{i,[k:k+V-1],[x-1:x+1],y}$. Thus, both D elements contribute to $Y_{i,[k:k+V-1],[x-1:x],y}$. As a result, as x increments, we can keep $Y_{i,[k:k+V-1],[x-1:x],y}$ in the registers. We only need to save $Y_{i,[k:k+V-1],x-2,y}$ to memory and load $Y_{i,[k:k+V-1],x+1,y}$ from memory. Consequently, each Y vector is only read and written once during a row sweep.



(a) When proceeding to the next D element during a row sweep, we do not store and then reload the output Y vectors affected by both the current and the next inputs. However, a naïve implementation requires transferring data between registers.



(b) Cyclic register renaming further avoids transferring data between registers.

Figure 1: Examples of how *SparseTrain* minimizes both memory access and moving data between registers during a row sweep when $R = 3$ and $O = 1$.

However, although the shared Y vectors can stay in the registers as x advances, a naïve implementation that statically uses registers according to the spatial order of the convolution still requires transferring the Y vectors from one register to another. For example, in Figure 1a, `zmm[0:2]` hold the Y vectors affected by a D element in the order from left (lower index in the W dimension) to right (higher index in the W dimension). As x increments, the Y vector in `zmm[1:2]` needs to be transferred to `zmm[0:1]`. Modern microarchitectures typically eliminate such register-to-register moves at the register allocation stage to bypass executing them in the back-end [10]. Nevertheless, the move instructions still consume front-end resources.

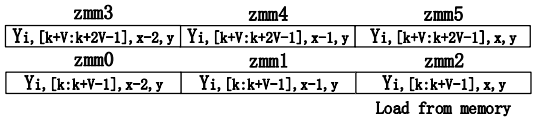
To avoid the move instructions, we devise a software scheme that simulates register renaming. As illustrated in Figure 1b, we use `zmm[0:2]` to hold the Y vectors. When working on $D_{i,c,x,y}$, `zmm0` holds $Y_{i,[k:k+V-1],x-2,y}$, `zmm1` holds $Y_{i,[k:k+V-1],x-1,y}$, and `zmm2` holds $Y_{i,[k:k+V-1],x,y}$. After moving on to $D_{i,c,x+1,y}$, `zmm0` proceeds to load $Y_{i,[k:k+V-1],x+1,y}$ while $Y_{i,[k:k+V-1],x-1,y}$ and $Y_{i,[k:k+V-1],x,y}$ are kept in their previous registers.

This scheme requires unrolling the row sweep loop, starting on Line 7. For large W, fully unrolling can lead to kernels larger than the instruction cache. Since the cyclic renaming repeats every R iterations, we instead unroll by a factor of R to limit code size.

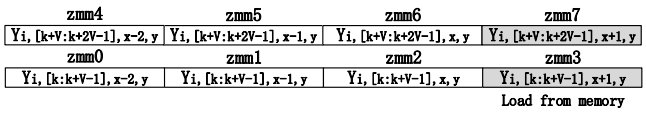
The number of registers used, how they are cyclically renamed, and the unrolling factor all depend on the parameters R and O. As a result, statically compiled code cannot implement this scheme. Hence, it is crucial to use JIT compilation.

Because R and V are fixed by the convolution configuration and the hardware, respectively, the only tunable parameter in $T = R \times Q/V$ is Q. As a result, the register budget is often underutilized. To see why, assume that we want Q to be a factor of the number of output channels K, so blocks have the same size. When $R = 5$, $V = 16$, and $K = 256$, which is a typical number of channels, a reasonable maximum value of Q is 64. As a result, $T = 20$. Recall that we have 32 vector registers in total, and we use 2 vector registers for other uses: one to hold an all-zero vector and the other to hold the broadcasted input D element. Therefore, 10 registers are unused.

In such cases, we use the spare registers to pipeline the load of the Y vector affected by the next D element. Consider the registers in Figure 2 that hold Y vectors when processing $D_{i,c,x,y}$. Figure 2a is the case without pipelining. With $R = 3$ and $O = 1$, we need 3 registers along the W dimension. Because we vectorize along the K dimension, with $Q = 32$ and $V = 16$, we need $Q/V = 2$ registers along the K dimension. Therefore, we allocate 6 registers in total. In this case, we load $Y_{i,[k:k+V-1],x,y}$ and $Y_{i,[k+V:k+2V-1],x,y}$ from memory. $D_{i,c,x,y}$ contributes to both of them.



(a) Without pipelining, 6 registers are used.



(b) With pipelining, 8 registers are used. The pipelined loads are marked with gray background.

Figure 2: Example allocations of the output buffer registers that hold Y vectors affected by $D_{i,c,x,y}$ when $R = 3$, $O = 1$, $Q = 32$, and $V = 16$.

Figure 2b is the case with pipelining. Because Q/V is unchanged in the example, we also need 2 registers along the K dimension. If we have 2 spare registers, we use them to preload Y vectors along the W dimension, i.e., we preload $Y_{i,[k:k+V-1],x+1,y}$ and $Y_{i,[k+V:k+2V-1],x+1,y}$. The next $D_{i,c,x+1,y}$ contributes to them, but the current $D_{i,c,x,y}$ does not. In this way, the VFMAs depend on loads from an *earlier* iteration so that the out-of-order hardware can dispatch the VFMAs sooner. Note that, with pipelining, the unroll factor of the row sweep loop becomes $R + 1$ instead of R.

We need $(R+1) \times Q/V$ registers as output buffers with pipelining or $R \times Q/V$ without it. Therefore, we want the number of output buffer registers to be maximized but no higher than the budget, which is 30 as discussed. At $K = 256$ and $V = 16$, the optimal values of Q for common values of the filter width R are shown in Table 3. The values of Q are 128 for $R = 1$ with pipelining, 128 for $R = 3$ without pipelining, and 64 for $R = 5$ with pipelining.

For $R = 1$, we found that the alternative of $Q = 256$ without pipelining is slower. This is because when processing each D element in a row sweep, we compute $R \times Q/V$ VFMAs and load Q/V number of Y vectors from memory. Thus, the compute to load ratio is R. When $R = 1$, the ratio is so low that pipelining provides substantial benefit by hiding the load latency.

3.2.4 Reducing Branch Mispredictions. As discussed, the optimal T is ≤ 30 on the target CPU. Under this constraint, the zero checking

Table 3: Optimal value of Q for $K = 256$ and $V = 16$ at different R.

R	Q	Pipelined?	# of output buffer registers	T
1	128	Yes	$16 = (R+1)Q/V$	$8 = RQ/V$
3	128	No	$24 = RQ/V$	$24 = RQ/V$
5	64	Yes	$24 = (R+1)Q/V$	$20 = RQ/V$

and skipping method in Lines 8-13 of Algorithm 2 may induce so many branch mispredictions that the code actually slows down. To address the issue, we transform a series of branches to a single loop and reduce the number of branches by a factor of V. With many fewer branches, we drastically reduce the overall misprediction penalty.

Algorithm 3 shows the method that can replace Lines 8-13 in Algorithm 2. First, we compare the input vector to zeros to generate a mask (Line 1, which maps to Line 8 in Algorithm 2). This is done with the `vcmpps` instruction on the target CPU. Then, we use `popcnt` (Line 2) to count the number of 1s in the mask, which represents the number of non-zero elements in the input vector. After that, the code loops this number of times as shown in Lines 3-13, where each loop iteration processes a non-zero element from the input vector.

Algorithm 3: Zero Checking for Branch Performance.

```
   input    : input pointer D, filter pointer G
   output   : register array Y
   constant : filter offset B
1  m[0:V-1] = vect_cmp_neq_zero(D[0:V-1]);
2  o = population_cnt(m[0:V-1]);
3  for i = 0 to o − 1 do
4    z = trailing_zero_cnt(m);
5    D += z; G += z * B;
6    for j = 0 to Q/V fully unrolled do
7      for k = 0 to R fully unrolled do
8        Y[j][k][0:V-1]+=broadcast(D[0])*G[j][k][0:V-1]
9      end
10   end
11   m = shift_right(m, z+1);
12   D += 1; G += B;
13 end
```

In each iteration, we first count the number of trailing zeros (`z`) in the mask with the `tzcnt` instruction (Line 4). Then, we advance the input pointer by `z`, to reach the next non-zero element in the input vector. We also advance the filter pointer such that it points to the filter elements corresponding to the given non-zero input element. Finally, we do the VFMAs.

We fully unroll the loop nest in Lines 6-10. The `Y[j][k][0:V-1]` vectors shown in the loop body are actually in the output buffer registers, which are allocated though the cyclic renaming scheme discussed earlier. Finally, we shift the mask to the right by `z+1` to reflect that we have finished processing the rightmost non-zero input element (Line 11), and also adjust the input and filter pointers accordingly (Line 12).

For readability, we omit some low-level optimizations in Algorithm 3. Specifically, we pipeline the vector compare instruction such that the vector mask for the next iteration is generated during the current iteration. In this way, we can overlap the compute from the current D element with the load of the next D element. We also manually schedule and pipeline the integer instructions in the loop body to minimize dependence stalls. Moreover, we use shifts and load effective address (`lea`) instructions to reduce the strength of the integer multiplications and the number of integer instructions. In the end, each loop iteration of Lines 3-13 only contains 8 cheap integer instructions plus the VFMAs.

3.2.5 Memory Access Optimization. We structured both the working sets and the loop nest carefully for high memory performance. First, we set the lowest dimension of the datasets to a channel tile of size V. On the target CPU, this is the `zmm` vector register size and the cache line size. Recall that we vectorize the computation along channels. Therefore, when the channel tile is aligned to a cache line boundary, vector instructions operate efficiently on a vector of channel data.

We have 3 working sets, with different behaviors: the input D, the filters G, and the output Y. D and Y have spatial locality in a row sweep. Each row element from them is loaded/stored only once per row sweep, and adjacent elements in a row are accessed consecutively. Such a streaming pattern benefits from hardware prefetching when we assign the second lowest dimension to the row dimension. We may also strategically software-prefetch the elements of the next row to the L2 cache when the line fill buffers (LFB) are not saturated.

In contrast, G has temporal locality in a row sweep. Since we compute partial results for $W \times Q$ output elements from $W \times V$ input elements in a row sweep, we access $Q \times V \times R$ filter elements repeatedly. With the R and Q values listed in Table 3, when $R = \{3, 5\}$, 24KB or 20KB of G elements are used per row sweep. Thus, on a machine with a 32KB L1-D cache, the next set of G elements needs to be loaded from the L2 or below when the input/output channels of focus change. To counter the issue, we block the minibatch dimension (N) with a tile size of M to reuse each G element M times, as in Lines 1 and 6 in Algorithm 2. The heuristic is that $M = 16$ is appropriate for most convolution configurations.

Layers such as ReLU, pooling, LRN, normalization, and batch concatenation can be efficiently implemented on the same layout that the convolutional layers use [11], so in most cases we do not need to transpose the activations between layers.

3.3 Backward Propagation by Input (BWI)

For a unit-stride convolution, BWI is virtually the same as FWD, with the exception that the filters are flipped. However, non-unit strides introduce some differences. Specifically, when applying the register usage optimization described in Sec. 3.2.3 with horizontal stride $O > 1$, in FWD, we load Q/V new Y vectors into the accumulator registers after we finish processing O vectors of D. However, in BWI, we load $O \times Q/V$ new $\partial L/\partial D$ vectors into the accumulator registers after we finish processing one $\partial L/\partial Y$ vector.

Also, in a FWD row sweep, some D elements may contribute to a number of Y vectors that is less than T due to the horizontal stride; however, in a BWI row sweep, except for the image boundaries,

an $\partial L/\partial Y$ element always contributes to T $\partial L/\partial D$ vectors. We generate the appropriate number of skippable VFMAs through JIT.

Finally, the unroll factor of the row sweep loop in FWD is $W \times O$; it is the least common multiple of W and O in BWI.

3.4 Backward Propagation by Weights (BWW)

Algorithm 4 is a naïve sparse algorithm for BWW. It checks for zeros in D. We can easily modify the algorithm to check for zeros in $\partial L/\partial Y$ instead, if we expect more sparsity in $\partial L/\partial Y$ of the target layer. In Algorithm 5, we improve on Algorithm 4 by applying output-parallelization and similar optimizations used in FWD and BWI, with some changes.

Algorithm 4: Naïve Vectorized Sparse BWW.

input : input D, output gradients dY
output : filter gradients dG

1 **for** $i = 0, c = 0, y = 0, x = 0$ **to** $N-1, C-1, H-1, W-1$ **do**
2 **if** $D_{i,c,x,y} \neq 0$ **then**
3 **for** $k = 0$ **to** $K - V$ **step** V **do**
4 **for** $u = 0, v = 0$ **to** $R-1, S-1$ **do**
5 $dG_{[k:k+V-1],c,u,v} =$
 $dG_{[k:k+V-1],c,u,v} + D_{i,c,x,y} \times dY_{i,[k:k+V-1],x-u,y-v}$;

Algorithm 5: Parallel Vectorized Sparse BWW.

input : input D, output gradients dY
output : filter gradients dG

1 **for** $i = 0$ **to** $N - V$ **step** V **do**
2 **for** $y = 0$ **to** $H - 1$ **do**
3 **for** $v = 0$ **to** $S - 1$ **in parallel do**
4 **for** $k = 0$ **to** $K - Q$ **step** Q **in parallel do**
5 **for** $c = 0$ **to** $C - 1$ **in parallel do**
6 **for** $x = 0$ **to** $W - 1$ **do**
7 $m_{[0:V-1]} = [d \neq 0$ **for** d **in** $D_{[i:i+V-1],c,x,y+v}]$;
8 **for** $i' = 0$ **to** $V - 1$ **do**
9 **if** $m_{i'}$ **is true then**
10 **for** $k' = k$ **to** $k + Q - V$ **step** V **do**
11 **for** $u = 0$ **to** $R - 1$ **do**
12 $dG_{[k':k'+V-1],c,u,v} = dG_{[k':k'+V-1],c,u,v} +$
 $D_{i+i',c,x,y+v} \times dY_{i+i',[k':k'+V-1],x-u,y}$;

In Algorithm 5, we vectorize the zero-checking along the minibatch dimension (N) instead of the channel dimension as in FWD and BWI, reflected in Line 7. This is because in BWW, the destination of the VFMA operation, $dG_{[k:k+V-1],c,u,v}$, changes as the input channel c changes. As a result, if we vectorize the zero-checking along the input channel dimension (C), we need to store the previous group of $dG_{[k:k+V-1],c,u,v}$ vectors to memory and load a new group before entering the loop starting at Line 10, and this frequent register spilling may harm performance significantly. Luckily, because $dG_{[k:k+V-1],c,u,v}$ is minibatch-invariant, all input elements from the vector $D_{[i:i+V-1],c,x,y+v}$ contribute to the same group of $dG_{[k:k+V-1],c,u,v}$ vectors. Therefore, vectorizing the zero-checking along the minibatch dimension avoids spilling the registers.

Due to the change in vectorization scheme, we transpose D such that the lowest dimension is a minibatch tile of size V. This allows us to load D vectors directly as opposed to gathering from locations apart.

In a row sweep, a core works on $R \times Q$ filter gradients. Because the total number of filter gradients is $R \times S \times K \times C$, the maximum parallelism becomes $S \times C \times K/Q$.

Since the set of filter gradient elements is constant during a row sweep, if we limit the number of filter gradient vectors being worked on, which is $T = R \times Q/V$, to the register budget, they can stay in the registers during the entire row sweep. Consequently, we do not apply the cyclic register load/store and renaming scheme described in Section 3.2.3. This also lifts the restriction on the unrolling factor for the row sweep loop so that it can be chosen freely.

Instead of loading the previous partial results of the ∂G vectors at the beginning of a row sweep, and storing the new partial results to memory at the end, we clear the accumulator registers at the beginning and store the VFMA results in them during a row sweep. At the end, we load the previous partial results and add them to the accumulator registers as the new partial results, and we immediately store them back to memory afterwards. Therefore, the filter gradient elements are only accessed twice in succession at the end. We also prefetch the filter gradient elements in software at the beginning. With this optimization, we do not need to tile the minibatch dimension to reuse the filter elements as described in Sec. 3.2.5.

The two multiplicand vectors of the VFMA instructions in BWW are the broadcasted input element $D_{i+i',c,x,y+v}$ in a vector register and the $\partial L/\partial Y$ vector $dY_{i+i',[k':k'+V-1],x-u,y}$ as a memory operand.

3.5 Generalization to Other Hardware

We implement *SparseTrain* with AVX-512's vector FMA and vector comparison instructions. Other ISAs beyond x86, such as ARM Neon [6], also support them. Nevertheless, the techniques are generalizable to ISAs without them. Without vector comparison, we may fall back to comparing each scalar element to zero. Note that we still need to compare a batch of scalar elements at once and then apply Algorithm 3 to combat branch misprediction. On machines without vector FMA, *SparseTrain* is actually more effective. This is because with scalar FMAs, the number of skippable instructions per zero input is much higher, which can more easily hide the branch misprediction penalty.

Our general idea is also applicable to GPUs. On CPUs, the main challenge is to reduce branch misprediction. On the other hand, on GPUs we need to minimize control divergence, which happens when threads in the same SIMD group, or *warp* in CUDA terms, take different paths in a control sequence. One possible solution is to let threads in a given warp compare the same input with zero simultaneously; therefore, all threads in the same warp may either issue or skip the computation. However, the method only works with general MAC computations, and not with hardware accelerated GEMM instructions such as Nvidia's Tensor Core MMA instructions, which compute a GEMM tile directly [24]. As a result, it may be hard for a GPU *SparseTrain* implementation to beat the Tensor Core accelerated GEMM. Nevertheless, the method can be useful on GPUs without a hardware GEMM accelerator (e.g., the integrated GPUs used for inference on edge devices [49]), or

when we desire higher precision than the one supported by the accelerator.

4 EXPERIMENTAL SETUP

We build *SparseTrain* as new kernels in *MKL-DNN* [21]. We use the *xbyak* JIT assembler [32] to generate the code. Because *TensorFlow* [1] uses MKL-DNN as the backend library on CPUs, we also integrate our new kernels into TensorFlow. We evaluate full network training/inference using TensorFlow with the *SparseTrain*-augmented *MKL-DNN*.

We use *MKL-DNN*'s direct convolution as the baseline, which we refer to as *direct*. *MKL-DNN* has three other implementations of convolution: (1) a method that first flattens the inputs with *im2col* and then applies a GEMM, (2) a vectorized *Winograd* convolution [27] for unit-stride 3×3 convolutions only, and (3) a special kernel that optimizes 1×1 convolutions. We compare *SparseTrain* to them when applicable.

We run our experiments on an Intel Skylake-X server with 6 cores. Each core has two AVX-512 vector units, 32KB L1 I- and D-caches, and a 1MB L2 cache. There is a non-inclusive 8.25MB shared L3 cache. We disable hyperthreading as well as dynamic frequency scaling. We run 6 threads in parallel.

We evaluate *SparseTrain* with VGG16 [43], ResNet-34, and ResNet-50 [17]. Batch Normalization (*BatchNorm*) [22] affects *SparseTrain*'s effectiveness because when the *conv-BatchNorm-ReLU* structure is present, $\partial L/\partial Y$ becomes dense. Since VGG16 does not employ the structure, *SparseTrain* benefits all of its FWD, BWI, and BWW. However, the two ResNet variants have the structure. Therefore, *SparseTrain* does not accelerate their BWI. Zhang et al. [54, 55] demonstrated that, with proper initialization and data augmentation, one can train ResNet without BatchNorm with marginal accuracy loss. Therefore, we also experiment with the BatchNorm-free ResNet-50, called *Fixup* ResNet-50.[3]

We first examine *SparseTrain*'s training performance with CIFAR-10 [25] as a proof of concept, and then evaluate with the larger ImageNet-1K [9] data set. Because CIFAR-10 is small, we train ResNet-34 with *SparseTrain* from end to end, and time all conv layers to obtain the training performance. However, training multiple DNNs with ImageNet on a small CPU server takes an unreasonable amount of time, so we adopt the following sampling-based method:

(1) We train a network from scratch on GPUs. During training, we checkpoint models at each epoch.
(2) For each epoch, we randomly sample 5 mini-batches. For each mini-batch, we run a training iteration with *SparseTrain* using the checkpoint model. We record the average execution time from the 5 samples as *SparseTrain*'s mean performance at the given epoch.
(3) We take the average sampled run time across all epochs as *SparseTrain*'s mean training performance.

We observe that the sparsity progression between adjacent epochs is smooth, and that the randomly-sampled mini-batches have low sparsity variations. Therefore, we are confident that the sampled performance faithfully approximates the overall training performance.

[3]We use the variant without the scalar bias between each ReLU and its subsequent conv layer.

Since *SparseTrain* also benefits inference, we use the trained models to evaluate *SparseTrain*'s inference performance.

Besides whole-network performance, we also assess *SparseTrain*'s performance on individual layers at various sparsity levels. For this, we generate synthetic inputs with random sparse patterns and experiment on all but the first conv layers of VGG and ResNet. We use a batch size of 16 during the experiments. Table 2 lists the layer configurations used.

5 EVALUATION

We first discuss the sparsity progression in end-to-end training, and then present *SparseTrain*'s speedup on whole-network training/inference at realistic sparsity. Next, we discuss *SparseTrain*'s performance on individual layers with different filter sizes, and finally evaluate our technique to mitigate branch misprediction.

5.1 Activation Sparsity in Training

Figure 3 presents the ReLU output sparsities at each epoch in the end-to-end training of VGG16, ResNet-34, ResNet-50, and Fixup ResNet-50 with ImageNet. Each plot shows numbered segments. Each segment is the output from a *conv-ReLU* cluster in VGG16/Fixup ResNet-50, or a *conv-BatchNorm-ReLU* cluster in ResNet-34/50. Within a segment, from left (in darker color) to right (in lighter color) are the sparsities from the first epoch to the last.

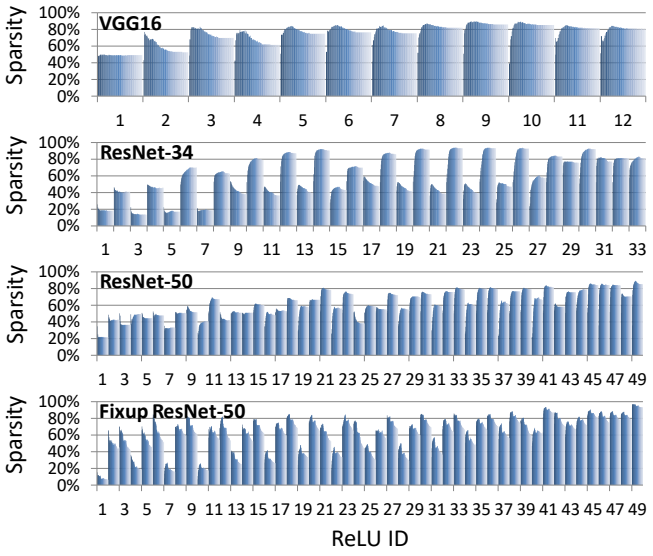


Figure 3: Measured ReLU sparsity in end-to-end training with ImageNet. Each numbered segment of the x-axis corresponds to a ReLU in the network. Within a segment, from left to right are the sparsities from the first epoch to the last.

The figure shows that the average sparsity of a layer typically ranges from 20% to 90%. Later layers generally have higher sparsity then earlier ones [38]. In addition, we also discover that, in the ResNet variants, the sparsity of adjacent layers fluctuates periodically. This is caused by the shortcut in each residual block, which adds positive biases before ReLU and, therefore, lowers the sparsity.

The *conv-ReLU* cluster and the *conv-BatchNorm-ReLU* cluster result in different sparse inputs to each training component. We list them in Table 4. Note that D of a convolutional layer is from the cluster before the layer, while $\partial L/\partial Y$ of a convolutional layer is from the cluster that contains the layer.

Table 4: The sparse input to different training components of conv layers.

	FWD	BWI	BWW
VGG16	D	$\partial L/\partial Y$	D and $\partial L/\partial Y$
ResNet-34	D	N/A	D
ResNet-50	D	N/A	D
Fixup ResNet-50	D	$\partial L/\partial Y$	D and $\partial L/\partial Y$

The table shows that, in vanilla ResNet-34/50, BWI has no sparse input at all due to BatchNorm, making *SparseTrain* ineffective. In this case, one may prefer to use a dense kernel instead. When we evaluate whole-network training, we substitute *SparseTrain* with *direct* for ResNet-34/50's BWI.

On the other hand, when BatchNorm is absent such as in VGG16 and Fixup ResNet-50, BWW's both inputs (D and $\partial L/\partial Y$) are sparse. Therefore, with heuristics or online profiling, one can configure *SparseTrain* to take advantage of the input that has a higher sparsity.

5.2 Whole-Network Performance

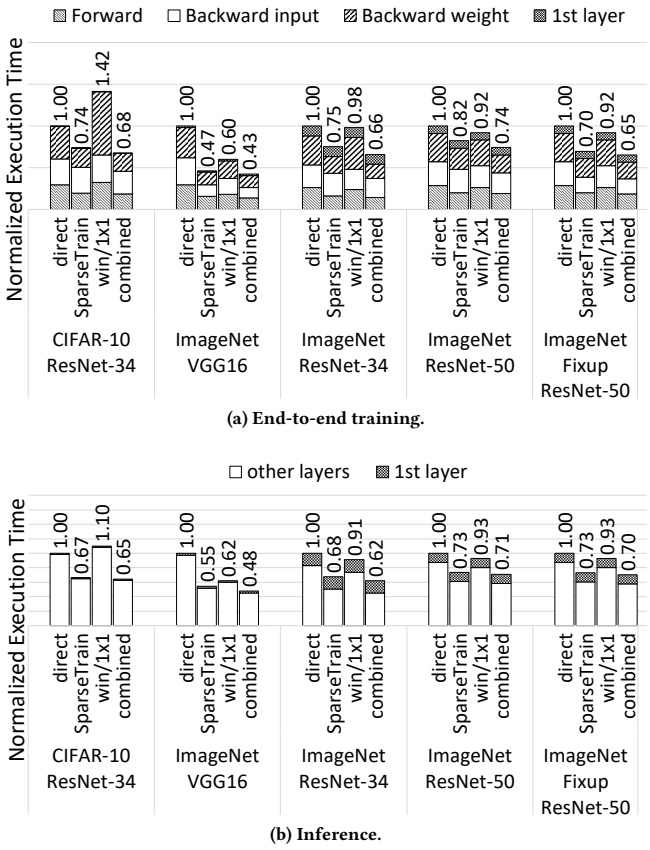
We now present *SparseTrain*'s whole-network performance. Figure 4 shows the end-to-end training (a) and inference (b) time of the convolutional layers with different networks and algorithms. For each network and algorithm, the execution time is normalized to that of *direct*. For training, we break the time into FWD, BWI, and BWW. Because *SparseTrain* is not applicable to the first layer in the network due to the input images often being zero-free, we show the first layer as a separate component.

In the figure, the *SparseTrain* bars correspond to using only *SparseTrain*, or in the case of the ResNet-34/50, using *SparseTrain* for FWD and BWW and *direct* for BWI. The *win/1x1* bars correspond to using the *Winograd* convolution or the optimized *1x1* kernel when possible; otherwise, we use *direct*. The *combined* bars combine the fastest algorithm of each layer. Finally, we find that the method using *im2col* plus GEMM described in Section 4 is consistently over 2x slower than *direct*, so we omit it in the figure.

In general, the CIFAR-10 and the ImageNet results are similar. *SparseTrain* achieves notable speedups on all networks. In contrast, *Winograd* performs well on VGG16 but worse on ResNet. Further, its performance with CIFAR-10 is much lower because it performs badly with small input width and height. Since CIFAR-10 has small input images (32×32), the input width and height are as low as 4 in the later layers.

Table 5 lists the speedups of the different algorithms over *direct*, both including and excluding the first layer. The first layer contributes 1-3% of the execution time of CIFAR-10 ResNet-34 and VGG16, but rises to 9-12% for the ImageNet ResNet variants. This is because the ImageNet ResNet variants have costly 7×7 first layers.

We can see that, when including the first layer, *SparseTrain* speeds up the training of the convolutional layers in the studied



(a) End-to-end training.

(b) Inference.

Figure 4: Breakdown of the execution time of all convolutional layers from different networks, normalized to the dense direct convolution. The training time of ResNet-34 on CIFAR-10 and all the inference times are measured from actual runs. The training times on ImageNet are estimated using the methodology described in Sec. 4.

networks by 1.28x-2.15x. By choosing the best algorithm for each layer, we can speed up training by 1.39x-2.35x. The speedup on VGG16 is notably higher than that on the ResNet variants because: (1) VGG16 has higher dynamic sparsity, and (2) VGG16 does not have BatchNorm, so *SparseTrain* is applicable to its BWI. Note that *SparseTrain* speeds up Fixup ResNet-50 by 1.45x instead of 1.28x on the original ResNet-50. The reason is also the absence of BatchNorm. Without including the first layer, the speedups of *SparseTrain* are 2.19x, 1.37x, 1.31x, and 1.51x for VGG16, ResNet-34, ResNet-50, and Fixup ResNet-50, respectively, with ImageNet.

For inference, when including the first layer, *SparseTrain* speeds up the conv layers in the studied networks by 1.36x-1.83x. The numbers increase to 1.41x-2.09x after choosing the best algorithm for each layer. Without including the first layer, the speedups of *SparseTrain* are 1.88x, 1.64x, 1.44x, and 1.44x for VGG16, ResNet-34, ResNet-50, and Fixup ResNet-50, respectively, with ImageNet. Overall, *SparseTrain* delivers good speedups over the state-of-the-art across networks for end-to-end training and inference.

Table 5: Speedup of the different algorithms over the dense direct convolution on all of the evaluated networks' convolutional layers at realistic sparsity levels.

			Including the first layer			Excluding the first layer		
			SparseTrain	win/1x1	combined	SparseTrain	win/1x1	combined
Training	CIFAR-10	ResNet-34	1.35x	0.71x	1.47x	1.36x	0.70x	1.48x
	ImageNet	VGG16	2.15x	1.66x	2.35x	2.19x	1.68x	2.40x
		ResNet-34	1.31x	0.99x	1.48x	1.37x	0.98x	1.58x
		ResNet-50	1.28x	1.08x	1.39x	1.31x	1.09x	1.44x
		Fixup ResNet-50	1.45x	1.08x	1.53x	1.51x	1.09x	1.62x
Inference	CIFAR-10	ResNet-34	1.50x	0.91x	1.55x	1.51x	0.91x	1.57x
	ImageNet	VGG16	1.83x	1.60x	2.09x	1.88x	1.63x	2.15x
		ResNet-34	1.48x	1.10x	1.61x	1.64x	1.12x	1.83x
		ResNet-50	1.36x	1.08x	1.41x	1.44x	1.09x	1.50x
		Fixup ResNet-50	1.36x	1.08x	1.43x	1.44x	1.09x	1.52x

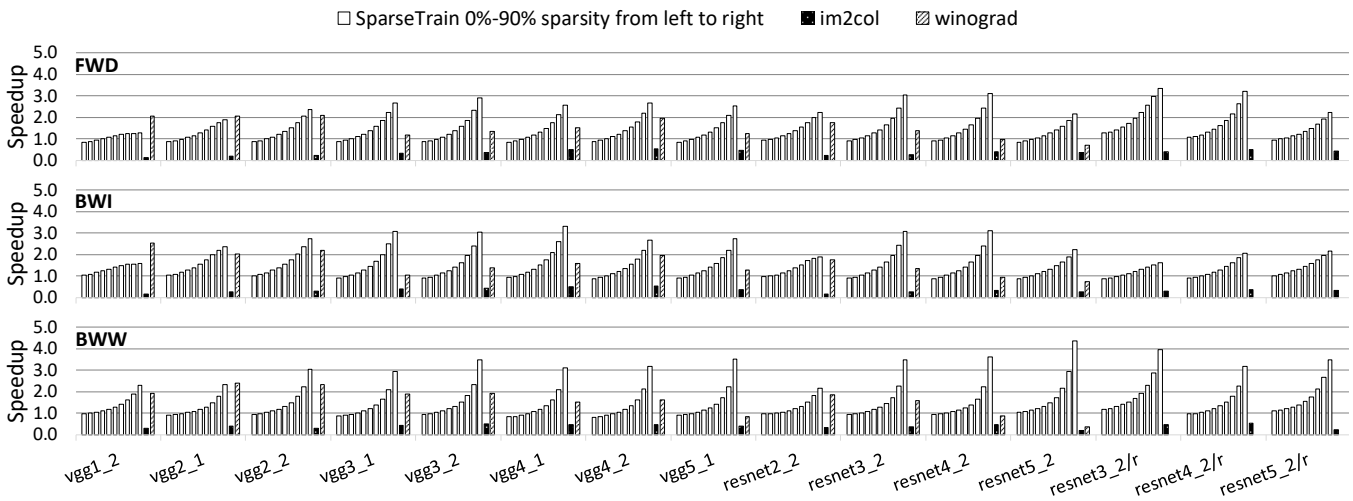


Figure 5: Speedup of the different algorithms over the dense direct convolution on the individual 3×3 layers at different sparsity levels.

5.3 3×3 Convolutional Layers

We now consider *SparseTrain*'s performance on individual convolutional layers with different filter sizes. We first discuss 3×3 ($R = S = 3$) filters, which have become more popular than larger filter sizes in recent years.

Figure 5 shows the speedup of *SparseTrain* at 0-90% sparsities, of *im2col*, and of *Winograd* over *direct*, for FWD, BWI, and BWW on the 3×3 layers from the evaluated networks. Table 6 lists the mean speedup at various sparsities.

The table shows that, at 0% sparsity (i.e., a dense input), *SparseTrain* reaches 92%-95% of *direct*'s performance on average, depending on the component. This indicates that the overhead to check for and exploit sparsity is low, and the loop order, as well as the tiling strategy of *SparseTrain* are effective.

The speedups of *SparseTrain* increase with the sparsity. On average, the sparsity cross-over point for *SparseTrain* to outperform *direct* is between 10%-20%, which is lower than the observed sparsity during training. At 50% sparsity, which is the expected value at the beginning of the training when the distribution of the weights is centered at 0, *SparseTrain* on average delivers a 1.30x-1.40x speedup. Typically, the later layers in a network have higher sparsity than the earlier layers. For the later layers, the sparsity reaches over 90% for VGG16 and ResNet-34, and over 80% for ResNet-50. At such levels, *SparseTrain* is on average over 2x faster than *direct*.

Table 6: Mean speedup at various sparsity for 3×3 layers.

	SparseTrain										*im2c.*	*win.*
	0%	10%	20%	30%	40%	50%	60%	70%	80%	90%		
FWD	0.92	0.96	1.04	1.13	1.24	1.38	1.56	1.79	2.11	2.48	0.33	1.45
BWI	0.93	0.98	1.06	1.15	1.26	1.40	1.58	1.81	2.10	2.45	0.31	1.48
BWW	0.95	0.98	1.03	1.10	1.18	1.30	1.48	1.76	2.23	3.15	0.37	1.44

im2col is always significantly slower than *direct*. Although GEMM on CPU is highly optimized, the flattening of the inputs through *im2col* incurs severe time and memory overhead.

Winograd is only applicable to unit-stride layers. For these layers, *Winograd* is on average 1.44x-1.48x faster than *direct*. However,

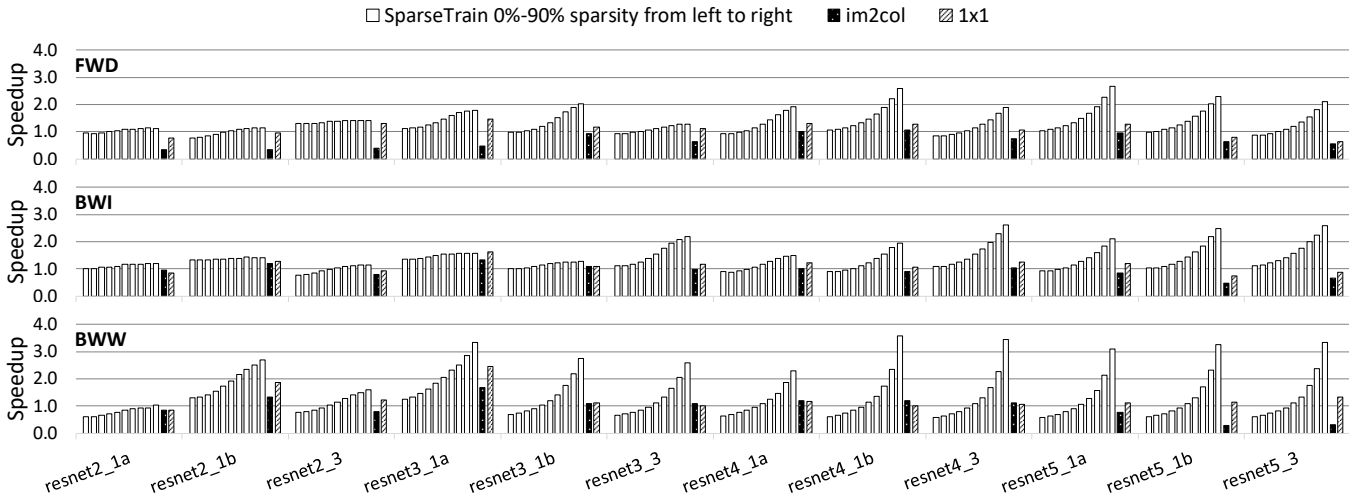


Figure 6: Speedup of the different algorithms over the dense direct convolution on the individual 1×1 layers at different sparsity levels.

because the *Winograd* algorithm transforms the problem to the "Winograd space," it has two drawbacks that are absent in *SparseTrain*. First, the transformation introduces numerical instability as the filter size increases, so its application is usually limited to 3×3 layers [48]. Second, it requires additional workspace memory.

SparseTrain performs better at later layers (e.g., *vgg5_1*), while *Winograd* dominates at earlier layers (e.g., *vgg1_2*). This is partly due to the increased sparsity at later layers; on average, it takes at least 50%-60% sparsity for *SparseTrain* to surpass *Winograd*. The other reason is that early layers have a smaller number of channels, which limits the number of skippable VFMAs per input element, and thus reduces the efficiency of *SparseTrain*. For example, both *vgg1_2* and *resnet2_2* have C and K of 64, giving us only 12 skippable VFMAs in *SparseTrain*. Overall, since *SparseTrain* and *Winograd* have different specialties, they can supplement each other.

SparseTrain for BWI delivers similar performance as for FWD with unit-stride. However, for stride-2 layers (layers with the /r suffix in Figure 5), BWI is slower than FWD. As discussed in Section 3.3, $\partial L/\partial D$ needs to be loaded O^2 times more rapidly during a row sweep in BWI than Y being loaded in FWD. Therefore, BWI suffers from cache bandwidth limitations.

5.4 1×1 Convolutional Layers

1×1 layers ($R = S = 1$) are widely used in ResNet-50's bottleneck blocks. They are unique amongst convolutions in that the spatial reuse of $R \times S$ is absent.

Figure 6 shows the speedup of *SparseTrain* at 0-90% sparsities, of *im2col*, and of the specialized *1x1* kernel over *direct*, for FWD, BWI, and BWW on the 1×1 layers from the evaluated networks. Table 7 lists the mean speedup at various sparsities.

SparseTrain exploits a convolution's high compute-to-memory ratio. However, the ratio for 1×1 layers is 9x lower than that for 3×3 layers with the same input/output/channel sizes. Thus, as we eliminate useless VFMAs, 1×1 layers may become bandwidth-bound sooner than 3×3 layers. Therefore, at high sparsity, *SparseTrain* is less effective on 1×1 layers than on 3×3 layers, only reaching 1.66x-2.04x speedups on average at 80% sparsity.

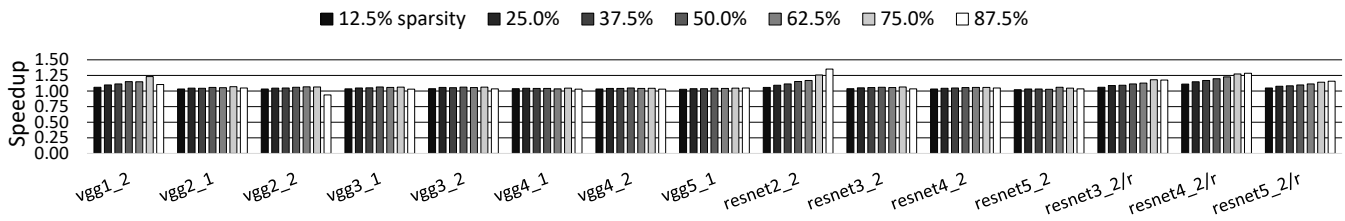
Table 7: Mean speedup at various sparsity for 1×1 layers.

	SparseTrain										*im2c.*	*1x1*
	0%	10%	20%	30%	40%	50%	60%	70%	80%	90%		
FWD	0.97	0.98	1.03	1.09	1.17	1.27	1.39	1.51	1.66	1.78	0.62	1.06
BWI	1.03	1.03	1.08	1.15	1.22	1.33	1.43	1.53	1.66	1.76	0.91	1.08
BWW	0.71	0.76	0.83	0.92	1.05	1.20	1.39	1.66	2.04	2.61	0.87	1.23

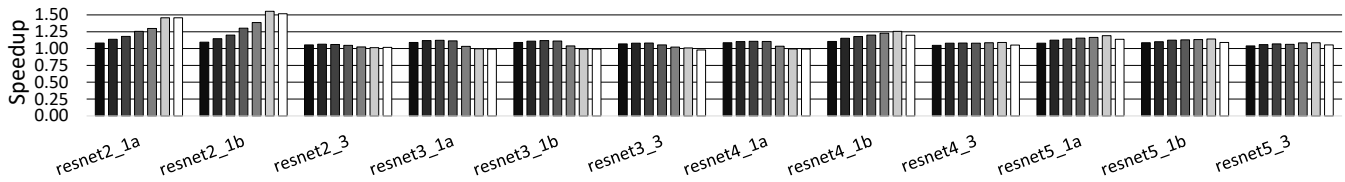
We also notice that BWW behaves differently than the other two components. At 0% sparsity, *SparseTrain*'s performance is on par with the *direct* for FWD and BWI. For BWW, though, *SparseTrain* only attains 71% of *direct*. However, at high sparsity, *SparseTrain*'s speedup is higher for BWW than for FWD and BWI.

The difference between BWW and FWD stems from two competing factors, both related to how BWW accesses $\partial L/\partial Y$ against how FWD accesses Y. First, BWW uses a different loop order, and in a row sweep touches V times more elements from $\partial L/\partial Y$ than FWD touches Y at 0% sparsity. Second, BWW reads $\partial L/\partial Y$ elements as a memory operand of a VFMA. When we skip a group of VFMAs, we also skip the access to the $\partial L/\partial Y$ elements. At high sparsity, we eliminate many such accesses. In contrast, FWD loads and stores Y elements using the cyclic register allocation scheme described in Sec. 3.2.3. Therefore, the Y elements are loaded and stored regardless of sparsity pattern. As a result, at low sparsity, BWW performs many more memory accesses, and at high sparsity, performs many fewer. This effect is less visible at 3×3 layers due to their higher compute-to-memory ratio; however, it is very obvious at 1×1 layers.

The fewer channels in earlier 1×1 layers hurts *SparseTrain* more than they do in earlier 3×3 layers due to the absence of spatial reuse. For example, *resnet2_1a* has 64 for C and K, resulting in only 4 VFMAs being skippable per zero-checking. Consequently, we can



(a) 3×3 **layers.**



(b) 1×1 **layers.**

Figure 7: Speedup over *SparseTrain* FWD when branch mispredictions in Algorithm 3 are eliminated.

hardly see any speedup from *SparseTrain* on earlier 1×1 layers. Nonetheless, we can still efficiently leverage the dynamic sparsity in later 1×1 layers.

As shown in Table 7, on average, the cross-point sparsity for *SparseTrain* to surpass the specialized *1x1* kernel is around 20% for FWD, at 0% for BWI, and around 40% for BWW.

In addition to 1×1 and 3×3 layers, we also experimented with several 5×5 layers from AlexNet [26] and GoogLeNet [46] and got even higher speedups. We omit the results due to lack of popularity of the 5×5 layers. Finally, we confirmed that *SparseTrain*'s execution time scales linearly with minibatch size N by experimenting with $N = \{32, 64\}$.

5.5 Mitigating Branch Misprediction Penalty

The unpredictable loop branch in Line 3 of Algorithm 3 accounts for most of the branches in *SparseTrain* because it is in the innermost loop. Moreover, the rest of the branches are all predictable. Therefore, it is crucial for Algorithm 3 to hide the branch misprediction penalty. To evaluate the performance of Algorithm 3, we design a study that quantifies the performance headroom that *SparseTrain* has if the branch in Line 3 was instead predictable.

In Algorithm 3, branch mispredictions stem from an unpredictable number of non-zero elements in an input channel vector (o in Line 2). In our study, we eliminate the misprediction at steady state by using special input data that has a fixed number n of non-zero elements per input channel vector. With this input, a local history predictor can easily predict the loop branch. In the experiments, we vary n from 1 to 15 (which is the value of $V - 1$). As a result, the sparsity of the input is $1 - n/V$ for a given n.

Figure 7 shows the speedup from eliminating branch mispredictions at steady state in Algorithm 3, at selected sparsity levels. The 3×3 layers are on the top, and the 1×1 layers are at the bottom. Each bar shows *SparseTrain*'s FWD execution time with a random input (*with* branch misprediction) over *SparseTrain*'s FWD execution time with the special input (*without* branch mispredictions), at different sparsity levels.

In most 3×3 layers (Figure 7a), *SparseTrain* sets the number of skippable VFMAs (T) to 24. As a result, the misprediction penalty is well hidden. Consequently, eliminating misprediction generally provides only up to 5% speedup.

There are, however, some exceptions. First, $vgg1\_2$ and $resnet2\_2$ both have a small output channel of $K = 64$. Therefore, *SparseTrain* can only set T to 12, which is insufficient to fully hide the misprediction penalty. Second, the stride-2 layers (layers with the /r suffix) have a variable T that is on average less than 24, exposing some of the penalty. As a result, eliminating misprediction speeds up these layers as sparsity increases, reaching up to 35% for $resnet2\_2$ at 87.5% sparsity.

The 1×1 layers (Figure 7b) have a lower T of ≤ 16 due to a lack of spatial reuse. The worst case is $resnet2\_1a$ and $resnet2\_1b$, whose T is only 4. For these layers, eliminating misprediction can yield over 50% speedup at high sparsity. Nonetheless, Algorithm 3 performs relatively well on other 1×1 layers, leaving only a small room for improvement.

In conclusion, while Algorithm 3 is less effective as T decreases, it performs well and close to the upper bound on most of the studied layers. Further reducing mispredictions in software may be hard. However, previous hardware proposals [41] could help: since the loop trip count is generated outside of the loop body, the count could be communicated to the branch predictor in hardware, completely eliminating branch mispredictions.

6 RELATED WORKS

Various works compress DNN models by eliminating redundant weights [14, 35]. Weight quantization [58, 59] sacrifices numerical precision to reduce model size. Structured sparsity [50] is more hardware-friendly, but it is inapplicable to training and does not exploit dynamic sparsity in the activation.

PruneTrain [29] prunes entire channels and reconfigures the model to a smaller dense form during training. *SparseTrain* is orthogonal to it. *SparseTrain* can further leverage the activation sparsity after PruneTrain reconfigures the model.

meProp [45] sparsifies the back propagation of LSTMs and MLPs by only propagating a small number of gradients in each pass. This reduces back propagation time. Yet, it does not affect the forward propagation, nor has it been tested on CNNs. Our work is orthogonal to it and can be applied in conjunction with it.

Several DNN accelerators exploit the sparsity in weights and/or activations. Cnvlutin [3] leverages activation sparsity to skip ineffectual computations. Eyeriss [7] clock-gates the hardware when there are zeros in the activation. It saves energy but not cycles. Cambricon-X [57] skips multiplications associated with pruned weights. EIE [13] exploits sparsity in both weights and activations of fully connected layers. SCNN [34] leverages sparsity in both weights and activations of conv layers. These accelerators are application specific, while our work targets general-purpose processors.

SparCE [40] skips ineffectual code blocks based on a sparse input. It annotates skippable code blocks in software and tests conditions in hardware. Therefore, it requires hardware-software co-design. Also, it mainly works on scalar code. We target high-performance SIMD CPUs and use software only.

ZCOMP [2] adds new instructions to load/store vectors from/to memory in a compressed form. It perfectly synergizes with *SparseTrain* because its reduction in memory traffic is proportional to *SparseTrain*'s reduction in compute intensity.

SAVE [12] proposes a vector unit for CPUs that exploits DNN sparsity in hardware. If one of the multiplicands of a VFMA's vector lane is zero, that lane is deemed ineffectual. SAVE combines effectual vector lanes from multiple VFMA instructions before issuing the compacted computation. If the whole VFMA instruction is ineffectual, SAVE still fetches and decodes it before skipping it; *SparseTrain* skips it entirely.

Apart from sparsity, algorithmic transformations are developed to speed-up convolution. Georganas et al. [11] and Zhang et al. [56] demonstrate that on CPUs, well-tuned direct convolution is much faster than the conventional *im2col*-based convolution. For larger filter sizes, the FFT-based convolution [31] accelerates the computation by operating in the frequency space. For smaller 3×3 layers, the Winograd algorithm [27] reduces computation by incorporating the Chinese Remainder theorem. However, the transformation introduces numerical instability with larger filter sizes and erases the dynamic sparsity in the activation.

Liu et al. [28] restore the activation sparsity with the Winograd convolution by applying ReLU to the activation after transforming to the Winograd space. However, their approach changes the network structure. In addition, their focus is to reduce the operation count for running DNN inference on mobile devices, and they do not target training or an efficient vectorized implementation.

7 CONCLUSION

SparseTrain is the first approach to exploit dynamic sparsity in software for DNN training on general-purpose CPUs. *SparseTrain* identifies zeros in a dense data representation and performs vectorized computation. It is applicable to all major components of training: forward propagation, backward propagation by inputs, and backward propagation by weights. We evaluate *SparseTrain* on a 6-core Intel Skylake-X server. In end-to-end training of VGG16, ResNet-34, and ResNet-50 with ImageNet, *SparseTrain* outperforms a highly-optimized direct convolution on the non-initial convolutional layers by 2.19x, 1.37x, and 1.31x, respectively. *SparseTrain* also benefits inference. It accelerates the non-initial convolutional layers of the aforementioned models by 1.88x, 1.64x, and 1.44x, respectively. Overall, *SparseTrain* is effective and opens up new research directions in speeding-up computations with modest sparsity.

ACKNOWLEDGMENTS

This work was partially supported by NSF under grant CCF 17-25734.

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/ Software available from tensorflow.org.

[2] Berkin Akin, Zeshan A Chishti, and Alaa R Alameldeen. 2019. ZCOMP: Reducing DNN Cross-Layer Memory Footprint Using Vector Extensions. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 126–138.

[3] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. 2016. Cnvlutin: Ineffectual-Neuron-Free Deep Neural Network Computing. In *Proceedings of the 43th Annual International Symposium on Computer Architecture*.

[4] Amazon. 2019. Amazon SageMaker ML Instance Types. https://aws.amazon.com/sagemaker/pricing/instance-types/.

[5] Dario Amodei, Rishita Anubhai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Jingdong Chen, Mike Chrzanowski, Adam Coates, Greg Diamos, Erich Elsen, Jesse Engel, Linxi Fan, Christopher Fougner, Tony Han, Awni Hannun, Billy Jun, Patrick LeGresley, Libby Lin, Sharan Narang, Andrew Ng, Sherjil Ozair, Ryan Prenger, Jonathan Raiman, Sanjeev Satheesh, David Seetapun, Shubho Sengupta, Yi Wang, Zhiqian Wang, Chong Wang, Bo Xiao, Dani Yogatama, Jun Zhan, and Zhenyao Zhu. 2015. Deep Speech 2: End-to-End Speech Recognition in English and Mandarin. arXiv:1512.02595 [cs.CL]

[6] ARM. 2016. ARM Compiler Version 5.06 armcc User Guide.

[7] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. 2016. Eyeriss: a Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. In *Proceedings of the 43th Annual International Symposium on Computer Architecture*.

[8] Dipankar Das, Sasikanth Avancha, Dheevatsa Mudigere, Karthikeyan Vaidynathan, Srinivas Sridharan, Dhiraj Kalamkar, Bharat Kaul, and Pradeep Dubey. 2016. Distributed Deep Learning using Synchronous Stochastic Gradient Descent. *arXiv preprint arXiv:1602.06709* (2016).

[9] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: a Large-Scale Hierarchical Image Database. In *2009 IEEE conference on computer vision and pattern recognition*. IEEE, 248–255.

[10] Agner Fog. 2019. The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers. *Copenhagen University College of Engineering* (2019), 02–29.

[11] Evangelos Georganas, Sasikanth Avancha, Kunal Banerjee, Dhiraj Kalamkar, Greg Henry, Hans Pabst, and Alexander Heinecke. 2018. Anatomy of High-Performance Deep Learning Convolutions on SIMD Architectures. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 830–841.

[12] Zhangxiaowen Gong, Houxiang Ji, Christopher W. Fletcher, Christopher J. Hughes, Sara Baghsorkhi, and Josep Torrellas. 2020. SAVE: Sparsity-Aware Vector Engine for Accelerating DNN Training and Inference on CPUs. In *Proceedings of the 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[13] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. 2016. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In *Proceedings of the 43th Annual International Symposium on Computer Architecture*.

[14] Song Han, Huizi Mao, and William J Dally. 2015. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. *arXiv preprint arXiv:1510.00149* (2015).

[15] Song Han, Jeff Pool, John Tran, and William Dally. 2015. Learning Both Weights and Connections for Efficient Neural Network. In *Advances in neural information*

processing systems. 1135–1143.

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

[17] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[18] Alexander Heinecke, Greg Henry, Maxwell Hutchinson, and Hans Pabst. 2016. LIBXSMM: Accelerating Small Matrix Multiplications by Runtime Code Generation. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 981–991.

[19] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *CoRR* abs/1704.04861 (2017). arXiv:1704.04861 http://arxiv.org/abs/1704.04861

[20] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. 2017. Densely Connected Convolutional Networks. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2017), 2261–2269.

[21] Intel. 2019. Intel(R) Math Kernel Library for Deep Neural Networks (Intel(R) MKL-DNN). https://github.com/intel/mkl-dnn.

[22] Sergey Ioffe and Christian Szegedy. 2015. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *arXiv preprint arXiv:1502.03167* (2015).

[23] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. 1–12.

[24] Andrew Kerr, Timmy Liu, Mostafa Hagog, Julien Demouth, and John Tran. 2019. Programming Tensor Cores: Natice Volta Tensor Cores with CULTLASS. https://developer.download.nvidia.cn/video/gputechconf/gtc/2019/presentation/s9593-cutensor-high-performance-tensor-operations-in-cuda-v2.pdf

[25] Alex Krizhevsky and Geoffrey Hinton. 2009. Learning Multiple Layers of Features from Tiny Images. (2009).

[26] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet Classification with Deep Convolutional Neural Networks. In *Advances in neural information processing systems*. 1097–1105.

[27] Andrew Lavin and Scott Gray. 2016. Fast Algorithms for Convolutional Neural Networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 4013–4021.

[28] Xingyu Liu, Jeff Pool, Song Han, and William J. Dally. 2017. Efficient Sparse-Winograd Convolutional Neural Networks. *CoRR* abs/1802.06367 (2017).

[29] Sangkug Lym, Esha Choukse, Siavash Zangeneh, Wei Wen, Sujay Sanghavi, and Mattan Erez. 2019. PruneTrain: Fast Neural Network Training by Dynamic Sparse Model Reconfiguration. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 36.

[30] Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. 2013. Rectifier Nonlinearities Improve Neural Network Acoustic Models. In *Proc. icml*, Vol. 30. 3.

[31] Michael Mathieu, Mikael Henaff, and Yann LeCun. 2013. Fast Training of Convolutional Networks Through FFTs. *arXiv preprint arXiv:1312.5851* (2013).

[32] Shigeo Mitsunari. 2019. Xbyak: JIT assembler for x86(IA32), x64(AMD64, x86-64) by C++. https://github.com/herumi/xbyak.

[33] Nvidia. 2015. GPU-Based Deep Learning Inference: A Performance and Power Analysis. https://www.nvidia.com/content/tegra/embedded-systems/pdf/jetson_tx1_whitepaper.pdf.

[34] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W Keckler, and William J Dally. 2017. SCNN: an Accelerator for Compressed-Sparse Convolutional Neural Networks. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*.

[35] Jongsoo Park, Sheng Li, Wei Wen, Ping Tak Peter Tang, Hai Li, Yiran Chen, and Pradeep Dubey. 2016. Faster CNNs with Direct Sparse Convolutions and Guided Pruning. In *Proceedings of the International Conference on Learning Representations*.

[36] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.). Curran Associates, Inc., 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[37] Alec Radford, Luke Metz, and Soumith Chintala. 2015. Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks. *arXiv preprint arXiv:1511.06434* (2015).

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

[39] Andres Rodriguez. 2017. Intel Processors for Deep Learning Training. https://software.intel.com/content/www/us/en/develop/articles/intel-processors-for-deep-learning-training.html

[40] Sanchari Sen, Shubham Jain, Swagath Venkataramani, and Anand Raghunathan. 2017. SparCE: Sparsity Aware General Purpose Core Extensions to Accelerate Deep Neural Networks. arXiv:1711.06315 [cs.DC]

[41] Rami Sheikh, James Tuck, and Eric Rotenberg. 2015. Control-Flow Decoupling: an Approach for Timely, Non-speculative Branching. *IEEE Trans. Comput.* 64, 8 (2015), 2182–2203.

[42] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. 2016. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature* 529, 7587 (Jan. 2016), 484–489. https://doi.org/10.1038/nature16961

[43] Karen Simonyan and Andrew Zisserman. 2014. Very Deep Convolutional Networks for Large-Scale Image Recognition. *arXiv preprint arXiv:1409.1556* (2014).

[44] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: a Simple Way to Prevent Neural Networks from Overfitting. *The journal of machine learning research* 15, 1 (2014), 1929–1958.

[45] Xu Sun, Xuancheng Ren, Shuming Ma, and Houfeng Wang. 2017. meProp: Sparsified Back Propagation for Accelerated Deep Learning with Reduced Overfitting. In *Proceedings of the 34th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 70)*. 3299–3308.

[46] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going Deeper with Convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 1–9.

[47] Dean Takahashi. 2018. Gadi Singer interview - How Intel designs processors in the AI era. https://venturebeat.com/2018/09/09/gadi-singer-interview-how-intel-designs-processors-in-the-ai-era/

[48] Kevin Vincent, Kevin Stephano, Michael Frumkin, Boris Ginsburg, and Julien Demouth. 2017. On Improving the Numerical Stability of Winograd Convolutions. In *International Conference on Learning Representations - Workshop Track*.

[49] Leyuan Wang, Zhi Chen, Yizhi Liu, Yao Wang, Lianmin Zheng, Mu Li, and Yida Wang. 2019. A Unified Optimization Approach for CNN Model Inference on Integrated GPUs. In *Proceedings of the 48th International Conference on Parallel Processing*. 1–10.

[50] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. 2016. Learning Structured Sparsity in Deep Neural Networks. *CoRR* abs/1608.03665 (2016). arXiv:1608.03665 http://arxiv.org/abs/1608.03665

[51] Carole-Jean Wu, David Brooks, Kevin Chen, Douglas Chen, Sy Choudhury, Marat Dukhan, Kim Hazelwood, Eldad Isaac, Yangqing Jia, Bill Jia, et al. 2019. Machine Learning at Facebook: Understanding Inference at the Edge. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 331–344.

[52] Koichi Yamada, Wei Li, and Pradeep Dubey. 2020. Intel's MLPerf Results Show Robust CPU-Based Training Performance For a Range of Workloads. https://www.intel.com/content/www/us/en/artificial-intelligence/posts/intels-mlperf-results.html

[53] Jiecao Yu, Andrew Lukefahr, David Palframan, Ganesh Dasika, Reetuparna Das, and Scott Mahlke. 2017. Scalpel: Customizing DNN Pruning to the Underlying Hardware Parallelism. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*.

[54] Hongyi Zhang, Moustapha Cisse, Yann N Dauphin, and David Lopez-Paz. 2017. mixup: Beyond Empirical Risk Minimization. *arXiv preprint arXiv:1710.09412* (2017).

[55] Hongyi Zhang, Yann N Dauphin, and Tengyu Ma. 2019. Fixup Initialization: Residual Learning Without Normalization. *arXiv preprint arXiv:1901.09321* (2019).

[56] Jiyuan Zhang, Franz Franchetti, and Tze Meng Low. 2018. High Performance Zero-Memory Overhead Direct Convolutions. *arXiv preprint arXiv:1809.10170* (2018).

[57] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. 2016. Cambricon-X: an Accelerator for Sparse Neural Networks. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1–12.

[58] Aojun Zhou, Anbang Yao, Yiwen Guo, Lin Xu, and Yurong Chen. 2017. Incremental Network Quantization: Towards Lossless CNNs with Low-Precision Weights. *arXiv preprint arXiv:1702.03044* (2017).

[59] Chenzhuo Zhu, Song Han, Huizi Mao, and William J Dally. 2016. Trained Ternary Quantization. *arXiv preprint arXiv:1612.01064* (2016).