Acorns: A Framework for Accelerating Deep Neural Networks with Input Sparsity

Xiao Dong*†, Lei Liu*, Peng Zhao*†, Guangli Li*†, Jiansong Li*†, Xueying Wang*† and Xiaobing Feng*†
*State Key Laboratory of Computer Architecture
Institute of Computing Technology
Chinese Academy of Sciences, Beijing, China
Email: {dongxiao, liulei, zhaopeng, liguangli, lijiansong, wangxueying, fxb}@ict.ac.cn
†University of Chinese Academy of Sciences, Beijing, China

***Abstract*—Deep neural networks have been employed in a broad range of applications, including face detection, natural language processing, and autonomous driving. Yet, the neural networks with the capability to tackle real-world problems are intrinsically expensive in computation, hindering the usage of these models. Sparsity in the input data of neural networks provides an optimizing opportunity. However, harnessing the potential performance improvement on modern CPU faces challenges raised by sparse computations of the neural network, such as cache-unfriendly memory accesses and efficient sparse kernel implementation.**

In this paper, we propose Acorns, a framework to *ac*celerate deep neural netw*o*rks with i*n*put *s*parsity. In Acorns, sparse input data is organized into our designed sparse data layout, which allows memory-friendly access for kernels in neural networks and opens the door for many performance-critical optimizations. Upon that, Acorns generates efficient sparse kernels for operators in neural networks from kernel templates, which combine directions that express specific optimizing transformations to be performed, and straightforward code that describes the computation.

Comprehensive evaluations demonstrate Acorns can outperform state-of-the-art baselines by significant speedups. On the real-world detection task in autonomous driving, Acorns demonstrates 1.8-$22.6\times$ performance improvement over baselines. Specifically, the generated programs achieve 1.8-$2.4\times$ speedups over Intel MKL-DNN, 3.0-$8.8\times$ speedups over TensorFlow, and 11.1-$13.2\times$ speedups over Intel MKL-Sparse.

***Keywords*-Deep Learning; Sparse; Optimization; Compiler;**

I. Introduction

Recent years have witnessed the remarkable advances in the development of deep learning models in many real-world problems, face recognition [1], [2], image retrieval [3], natural language processing [4], and autonomous driving [5], just to name a few. While these models are powerful enough to extract knowledge from massive data, they pose significant challenges in the computing efficiency when deployed to carry out inference. A typical neural network [6] requires billions of floating-point operations to classify an image, and more challenging problems, like the detection in autonomous driving, could bring several times more computations. Moreover, the advances in the capability of deep models heavily rely on the increase of model size, which directly translates into increasing computational demands and makes it more challenging and urgent to achieve fast inference. Considering the ubiquity in a broad spectrum of devices and good programmability, CPU plays an important role in the neural network workloads, especially for the inference phase [7].

Sparsity provides the opportunity to improve the inference performance, as redundant arithmetic operations can be safely skipped, and the memory traffic can be reduced. Actually, the input data is *naturally sparse* in many real-world tasks, such as the LiDAR (Light Detection And Ranging) detection [5], [8] in autonomous driving, face detection [2], character recognition [9], and image reconstruction and editing [10]. In these tasks, only specific regions in input images are valid. However, harnessing the potential improvement faces several challenges. First, irregular and data-dependent memory access pattern makes it hard to exploit the locality, leading to insufficient cache utilization. Secondly, implementing efficient sparse operators in neural networks can be complex. The sparse data is usually compressed and only valid elements are kept with indexing information, restricting available ways of the kernels to operate over the sparse data. Finally, although ignoring the input sparsity, existing frameworks [11]–[14] leverage the vendor libraries, such as Intel MKL-DNN [15] and NNPACK [16], to implement computationally expensive kernels. These libraries are elaborately optimized and tuned by experts to approach the peak performance [15], and thus are hard to outperform.

In this paper, we focus on exploiting the input sparsity to accelerate the inference of deep neural networks. Several techniques are proposed and work collaboratively to overcome the challenges. Based on the observation to the sparsity structure of the sparse inputs, an efficient data layout is designed for the sparse data in neural networks. The data layout provides cache-friendly access to sparse data for neural network kernels and helps to expose plenty of performance-critical optimization opportunities that can be easily exploited. With respect to operators, we design a generating method to produce efficient sparse kernels. We define kernel templates composed of code to describe the computation and optimizing directions to express specific transformations to be performed by the kernel code generator. Above the operator level, inter-operator optimizations are employed to simplify the end-to-end inference. All of the

978-1-7281-3613-4/19/$31.00 ©2019 IEEE
DOI 10.1109/PACT.2019.00022

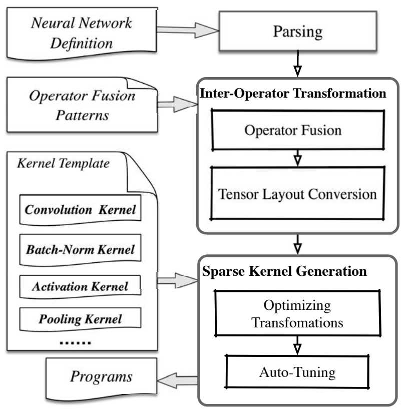


Figure 1. The workflow of Acorns

above techniques are integrated into a framework, Acorns, to achieve efficient inference of neural networks with input sparsity.

The workflow of Acorns is displayed in Fig. 1. Specifically, Acorns starts with taking the network topology as input and builds a computation graph consisting of the operators in the network. Before diving into each operator, Acorns performs two inter-operator transformations. The *operator fusion* finds and merges operators that can be fused together, and the *sparse tensor layout conversion* traces the intermediate tensors and ensures they match the requirement of tensors' data layouts when to be consumed. For operators in the transformed graph, Acorns performs specific transformations according to the optimizing directions on the code in templates and auto-tuning may be used to find the best configuration. The final generated program can achieve efficient inference of the neural network with sparse inputs.

To summarize, we make the following contributions in this paper:

- We design a new data layout for sparse input data in deep neural networks, which enables cache-friendly data accesses and helps to expose plenty of opportunities of performance-critical optimizations.
- We design a method to generate efficient sparse kernels in deep neural networks. The computing code and optimizing directions are combined into kernel templates, on which the kernel code generator performs corresponding optimizing transformations and tuning.
- We orchestrate all ingredients into an end-to-end framework to achieve efficient deep neural network inference with sparse input data.
- We conduct comprehensive experiments to evaluate the capability of Acorns and compare it with other state-of-the-art methods. Experiment results of the LiDAR-based detection demonstrate significant speedups over all baseline methods. Compared with existing sparse inference methods, Acorns delivers 8.9-13.2$\times$ performance improvement. As for dense inference methods, Acorns achieves 1.8-2.4$\times$ speedups over Intel MKL-DNN [15], 3.6-8.8$\times$ speedups over TensorFlow [11], and 9.8-16.8$\times$ speedups over NNPACK [16].

The rest of this paper is organized as follows. We first introduce necessary background in Section II. Section III overviews Acorns by an example. In Section IV, we introduce each part of Acorns in detail. The experimental evaluations are presented in Section V, followed by the comparison to related work in Section VI. Section VII summarizes this paper and introduces future work.

II. Background

In this section, we first provide a brief introduction to the key concepts about neural networks and notations we use in this paper. Then, we introduce typical tasks with sparse input data.

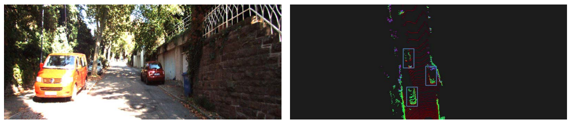
A. Neural Networks

A neural network is composed of a series of connected operators, with each one applying specific type of computation to the inputs from its dependent operators and passing the results to the following operators. Typical operators include convolution [17], batch normalization [18], non-linear activation [6], and pooling [6]. Among these operators, convolution is the focus of performance optimization, as it usually dominates the inference latency and is more complicated to optimize than others (e.g., matrix multiplications). A neural network containing convolutions is called convolutional neural network (CNN), which is one of the most important deep learning models and has been widely used in many domains, such as computer vision and natural language processing.

B. Tensors

The inputs and outputs of operators in neural networks are usually represented by tensors. A tensor can be viewed as a multidimensional array. In inference, the input and output tensors usually have three dimensions, the channel dimension (C) and two spatial dimensions, height (H) and width (W). Throughout this paper, we denote input, output and weight tensors with I, O and W, respectively. Without causing confusion, C, H, W are also used to represent the size of corresponding dimension.

The data layout of tensor can be expressed using the combination of dimensions, such as 'HWC' (channel dimension is the fastest varying dimension in memory). In the sparse situation, to save storage space, only non-zero values are kept and another data structure is built to store their indices. Popular sparse matrix formats such as compressed sparse row (CSR) [19], compressed sparse column (CSC) and coordinate (COO) are common examples. Only a particular set of access patterns can be efficiently supported by these data layouts. Moreover, as these formats are designed for



(a) RGB image (b) bird-eye-view LiDAR image

Figure 2. Example images from KITTI dataset. The RGB image captured by camera is on the left, and the corresponding bird's eye view image is on the right. The images are resized for displaying.

general sparse matrices, domain-specific sparsity structure cannot be captured by them. We introduce our designed data layout for sparse tenors in section IV-A.

C. Operators in Neural Network

The convolution operator convolves input feature map tensor with weight tensor, and produces new feature map tensor as output. The weight tensor consists of 4 dimensions: the kernel dimension K, the channel dimension C, the kernel height dimension R, and the kernel width dimension S. Convolution is equivalent to matrix multiplication (mm) when both R and S are equal to 1 and we name it non-spatial convolution. For spatial convolutions, common practice [11], [12], [20] first transforms the input tensor into a matrix and utilizes the mm kernel to compute the convolution result. Recently, direct convolution with careful optimizations plus jit (just-in-time) compilation shows excellent performance on Intel CPU [15], [21].

Pooling [6] divides the H and W dimensions into small 2D blocks and performs reduction within each block by picking up the maximum or computing the average. Activation (e.g., ReLU [6]), batch normalization (bn) [18] and scale [18] are all element-wise operators. Activation performs non-linear transformation while batch normalization and scale perform linear scaling with different coefficients for values in different channels. We refer readers to corresponding papers for more details.

D. Applications with Sparse Input

In this section, we introduce two representative and important tasks with sparse inputs, the LiDAR-based detection in autonomous driving and face detection.

1) LiDAR-based Detection: In autonomous driving, a car relies on the cooperation of multiple devices to sense the surrounding, such as cameras, GPS, and LiDAR sensors [22]. LiDAR-based perception is essential due to its ability to perceive distances accurately and the robustness to the lighting change. The sensor emits laser pulses and measures the distance to surrounding obstacles through the time taken by the pulse to reflect off the encountered object and return to the sensor. By drawing a point in a 3D coordination system where at least one pulse encounters object, a point cloud can be constructed to provide a good representation of the positions and shapes of surrounding objects, like pedestrians and other vehicles.

An effective way to process the point cloud is projecting it from the bird's eye view (the top view) and feeding the result images into deep learning detection algorithms to recognize objects of interests [5], [8]. These images are usually very sparse because only the surfaces of surrounding objects correspond to non-zero values. The sparsity can reach 75%-90% in real-world LiDAR-based detection [8], [23]. Fig. 2 displays example images.

2) Face detection: Face detection algorithms employ CNNs to check if there are faces in given images and output the locations of found ones. To reduce the risk of wrongly recognizing background as faces, some light-weight algorithms [2] are used to direct the attention of CNNs to predicted regions that are likely to have faces and ignore the left part of the image. The prediction result is expressed as a spatial mask and will be combined with the original image to serve as the input to following CNNs. As locations marked as 0 will be ignored by CNN, these values can be treated as 0 without affecting the final result.

III. Overview of Acorns

In this section, we use an example to give a brief introduction to how Acorns implements efficient inference of neural networks with sparse inputs. Subsequent section describe each part of Acorns in more detail.

Fig. 3 shows the process of generating the program for the example neural network, which is extracted from DenseNet [24]. Each step is shown in a block. First, Acorns takes the neural network topology (nn.prototxt) as input and parses it to construct a computation graph (Block 1). In the graph, each operator of the neural network is represented by a node, and edges represent tensors, including the input and output tensors of the neural network and the intermediate tensors that connect operators. The graph of the example network contains five operators.

The graph abstracts the neural network at the operator level and can serve as an appropriate representation to perform inter-operator transformations. Acorns first performs the *operator fusion* transformation to simplify the graph. In this transformation, Acorns traverses the graph and tries to find operator sequence that can be fused legally. In Block 1, Acorns finds the batch normalization, scale and ReLU operators can be fused (shown in the red box) and replaces them with a new operator having the same functionality. The simplified graph is shown in Block 2. The performance benefits from operator fusion since the memory traffic between fused operators is eliminated and the operational intensity is increased. In the original graph shown in Block 1, each of these three operators loads its input tensor from memory and writes the result tensor back after performing simple computations (linear scaling and

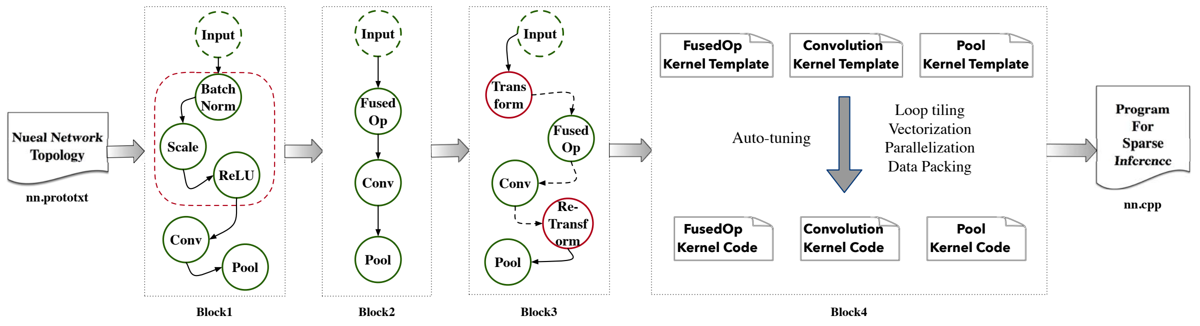
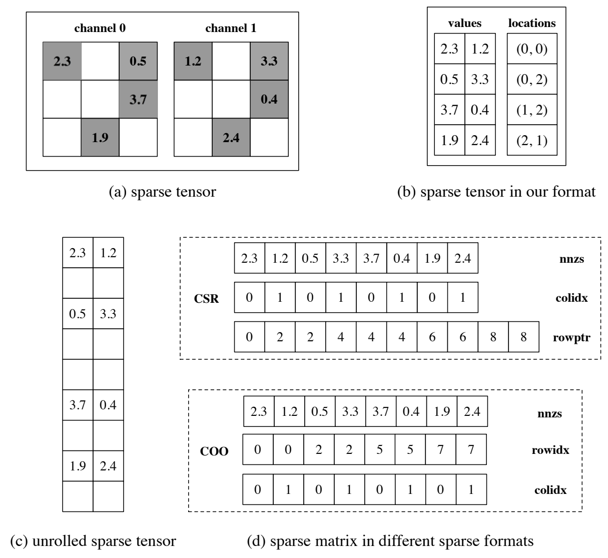


Figure 3. An example of the workflow of Acorns. The dashed input node represents a dummy operator feeding the network with inputs.

comparing). Operator fusion helps to reduce the time of loading and writing from three to one.

The following inter-operator transformation is *sparse tensor layout conversion*. For efficiency, some types of operators have specific requirements on the data layout of input tensors to avoid poor locality. Acorns ensures the requirements are satisfied by tracing the data layouts of tensors in the traversal over the graph and inserting layout conversion operators when necessary. At the beginning, we know the input tensor of the network is sparse and stored in the original dense tensor layout. Following edges, Acorns checks if it is compatible with the next operator. When mismatch happens, a data layout transformation operator is inserted. In Block 3, Acorns inserts a transformation operator (the red transform node) before the batch normalization to construct sparse tensors in our data layout, and appends a reverse transformation operator (the red re-transform node) after the convolution to match the pooling's preference for the original data layout.

The graph representation is lowered in the following *kernel code generation* step to generate efficient sparse kernels for operators in the transformed graph. We define kernel template for each type of operator. The template consists of straightforward computation code annotated with optimizing directions that express which transformations should be applied to the code. The optimizing directions supported by Acorns cover transformations that are recognized to be important for sparse kernels of neural networks, including loop tiling, vectorization, data packing and multithreading. The fused op in Block 3 performs series of element-wise operations and benefits from the multithreading parallelization. The sparse convolution is more complex (shown in Algorithm 3). In addition to parallelization, It contains *tiling* directions for blocking the loop nest to exploit the data reuse in input and weight tensors, and *vectorizing* direction for using the vector instruction FMA (fused multiply-add) to increase the throughput of arithmetic operations. Besides performing the code transformations according to the optimizing directions, Acorns employs auto-tuning to find the best tiling size. The generated kernels are combined to form



(a) sparse tensor

(b) sparse tensor in our format

(c) unrolled sparse tensor

(d) sparse matrix in different sparse formats

Figure 4. Sparse tensor in different formats

the final program to perform the inference of the neural network.

IV. Implementation of Acorns

A. Sparse Data Layout

The design of data layout for sparse tensors should take several requirements into account. Firstly, as operators' sparse input tensors are organized in this layout, it should allow operators to operate on the sparse tensors in a cache-friendly way. Secondly, it should support most of operators involved in neural networks, otherwise layout transformations will be frequently inserted into neural networks, resulting additional overhead. Thirdly, the layout transformations from and to the original layout should be efficient to avoid the performance gain is overwhelmed by the overhead of necessary transformations.

Based on the understanding of the sparsity structure of sparse input data to neural networks and the computing patterns of operators, we design an efficient sparse tensor layout. We make use of an important characteristic of the non-zero values' distribution in the input data, named as

Algorithm 1 Layout Transformation

```
 1: procedure TRANSFORM
 2:   for $(h, w) \leftarrow (0, 0)$ to $(H, W)$ do
 3:     if $I[h, w, 0] \neq 0$ then
 4:       $nnz += 1$
 5:       $values[nnz, :] = I[h, w, :]$
 6:       $locations[nnz, :] = [h, w]$
 7:     end if
 8:   end for
 9: end procedure

10: procedure REVERSE-TRANSFORM
11:   for $n \leftarrow 0$ to $nnz$ do
12:     $h, w \leftarrow locations[n, :]$
13:     $output[h, w, :] = values[n, :]$
14:   end for
15: end procedure
```

channel consistency, which means whether a given spatial location (h, w) is valid is consistent for all C values along the channel dimension. The channel consistency exists in the inputs of all tasks mentioned in Section II-D. Moreover, the channel consistency can be maintained for most types of operators in neural networks, such as convolution, pooling, batch normalization, scale and activation, which ensures the results of these operators are channel-consistent as long as the inputs are. Fig. 4 displays an example of a sparse tensor in our data layout. Assuming there are N non-zero spatial locations, we can know the total number of non-zero values is $N * C$, and the non-zeros can be stored in a dense matrix. Determining the storing order of the non-zeros requires the understanding of the memory access patterns of kernels. We arrange the channel dimension inside the merged spatial dimension, forming the dense 'NC' matrix named *values*. This ensures computationally expensive operators can access C values in a contiguous memory space. As for the location information, we store it in a $N * 2$ matrix named *locations* with each row recording the spatial location (h, w) for the corresponding C values.

Fig. 4 shows the example of representing the same sparse tensor in CSR and COO formats. To use sparse matrix formats, we first merge the 'height' and 'width' dimensions and construct a sparse matrix (Fig. 4c). As these formats are designed for general sparse matrices, they do not realize the channel-consistency in the sparse neural network domain, thus they ignore the location-sharing opportunity along the channel dimension, which results in larger memory footprint and reduces the operational intensity. COO stores a pair of row and column indices for each non-zero value, making the locations $C\times$ larger. CSR (CSC) records the column (row) indices for each non-zero value and encodes the row (column) indices in an extra array. Besides, all of these formats encode the locations in the sparse matrix, hence extra computation is required in kernels to restore the original location information of the tensor. Graham [25] proposed a hashtable-based representation. The hashtable serves as a rule book by recording which locations in the output tensor are related for each non-zero input location. However, it requires breaking convolution into several matrix multiplications, wasting the $R \times S$ folds of data reuse. Besides, the construction cost of the rule book is not trivial (See section V-B).

Our sparse tensor layout can support most of operators to access both *values* and *locations* efficiently. The overheads of transformations to this layout (transform) and back to the original layout (reverse-transform) are quite low. Algorithm 1 displays the two procedures.

Algorithm 2 Sparse Tensor Layout Conversion

```
 1: for $op$ in $G.operators$ do // topological order
 2:   for $input$ in $op.inputs$ do
 3:     if $input.layout \neq op.requiredLayout[input]$ then
 4:       $G.add\_transformation(op, input)$
 5:     end if
 6:   end for
 7:   for $out$ in $op.outputs$ do
 8:     $out.layout \leftarrow op.outputLayout[out]$
 9:   end for
10: end for
```

B. Inter-Operator Optimization

Acorns performs two transformations on the computation graph of neural network: *operator fusion* and *sparse tensor layout conversion*.

The operator fusion transformation targets to simplify the graph by finding consecutive operators that can be merged and replacing them with a new operator with the same functionality. This procedure is implemented in a traversal of the graph. Based on the understanding of operators' computation, we recognize common patterns of operators that can be fused to facilitate Acorns to find them. Each pattern consists of the types of those operators and their corresponding equivalent operator. During traversing the computation graph, Acorns compares the types of visited operators with defined patterns and performs the replacement when finding a match. Common patterns involves convolution and matrix multiplication, followed by normalization and activations, such as 'convolution-bn-scale-relu', 'mm-bn-scale' and 'bn-scale-relu'. Although possible combination of operators grows exponentially with operator types, most of them are incompatible with the design principles of neural networks and therefore the required patterns are limited.

Different types of operators may have different preferences for the data layouts of the input tensors due to their own data access patterns in computation. For examples, pooling desires efficient access to spatially neighboring values, which is ill-suited to spatially sparse tensors but can be well supported by the original tensor layout. In the sparse tensor layout conversion phase, Acorns needs to ensure the input tensors' layouts match the desired ones of corresponding operators to avoid performance degradation. Since the input tensors of one operator are produced by its dependent operators as outputs, we need to know the data layouts of each operator's output tensors.

Specifically, Acorns defines a pair of attributes for each type of operator. The *required layout* describes the preferred layouts of the input tensors, and possible values are 'original' for the original tensor layout, and 'sparse' for our sparse tensor layout. The other attribute *output layout* expresses the layout of each output tensor. With the two attributes, Acorns performs the data layout conversion via a traversal of the graph. Algorithm 2 displays the procedure. Starting from the first operator of the graph, Acorns records the layout of its output tensors. Then, following the dependency between operators, Acorns picks out operators whose input tensors are ready and checks whether layout transformation is required. Our designed sparse tensor layout is preferred by most of operators. Exceptions include pooling which prefers the original layout for both the input and output tensors, and spatial convolution which generates output tensor in the original layout. The overhead of layout transformations is evaluated in Section V-D.

C. Sparse Kernels

Based on the proposed sparse tensor layout, we design the kernels for common operators involved in neural networks. For element-wise operators, their kernels are quite straightforward and can be implemented in the traversal of the $values$ matrix. We do not list them due to the space limit.

Convolution usually dominates the end-to-end inference performance and can be viewed as a generalized matrix multiplication. We show the sparse convolution kernel in Algorithm 3. The basic idea is to traverse the rows in the matrix $values$ and multiply the C non-zeros in a row and the weight tensor. The result is added to the output tensor O. The original layout is used for the output tensor to determine the index for each output location ($outh$, $outw$) directly, and a mask matrix M records which locations are valid in the output tensor. Non-spatial convolution is equivalent to matrix multiplication and the result can be stored in the sparse layout directly.

In the sparse convolution kernel based on the designed data layout, plenty of optimizing chances are exposed to Acorns. There are ample data reuse in the computation. The C values in one row are used $R * S * K$ times by the weight tensor, and the weight tensor is also reused by different rows. As the non-zeros in our data layout are stored in a dense matrix with fixed shape, loop tiling and tuning can be easily used to enhance the data locality. Moreover, the core code at Line 13 computes the inner product of two vectors, which is suitable for vectorization. We introduce how these opportunities are captured by the optimizing directions in kernel templates in detail in the following section.

Algorithm 3 Sparse Convolution Kernel Template

```
 1: #pragma tiling
 2: #pragma parallelization
 3: for $i \leftarrow 0$ to $N$ do
 4:   $h, w \leftarrow I.locations[i, :]$
 5:   #pragma tiling
 6:   for $k \leftarrow$ to $K$ do
 7:     for $(r, s) \leftarrow (0, 0)$ to $(R, S)$ do
 8:       compute output spatial location $outh, outw$
 9:       $M[outh, outw] \leftarrow 1$
10:       #pragma tiling
11:       #pragma vectorizing
12:       for $c \leftarrow 0$ to $C$ do
13:         $sum += I.values[i, c] * W[k, r, s, c]$
14:       end for
15:       $O[outh, outw, k] += sum$
16:     end for
17:   end for
18: end for
```

D. Kernel Code Optimization

To harness the potential performance improvement, we need to exploit the optimizing opportunities and generate high-performance kernels for sparse operators. Implementing efficient sparse kernels is quite complex, and we show existing methods fail to achieve it in Section V. Compiler itself has many built-in optimizing transformations, but it is often restricted by the code complexity and the inability to prove the optimizations' validity [26]. Experts can write high-performance code based on the understanding of the computation and the knowledge of computer architecture, while this method takes significant engineering efforts, and achieving close-to-peak performance may still needs tuning.

Acorns adopts a template-based generating method to generate efficient kernels with the aid of the domain-specific optimizing knowledge. We define kernel templates that consist of straightforward code describing the computation and the optimizing directions expressing which transformations to be performed on the annotated code. Acorns will carry out specific transformations on the code automatically based on the optimizing directions. We introduce each optimization in detail next.

1) Loop tiling: Tiling [27] is a well-known optimization technique that transforms nested loops to enhance the data locality. It partitions the original loop space into small blocks and reorders the execution sequence such that data in cache and registers can be reused, hence reducing the number of references falling to the main memory and the data access latency.

Loops with *tiling* directions will be blocked automatically by Acorns. As the tiling size shows a big impact on the performance [28], [29] and is dependent on both the data access patterns and the hardware characteristics [29], [30], for flexibility and performance, Acorns takes the responsibility of finding the best tiling configuration for given loop nests and platform. It achieves this target through an auto-tuning method. Details are introduced in Section IV-D3.

Tiling directions are applied in kernels with data reuse chances, such as matrix multiplication and convolution. For example, we annotate the loops of N, K, and C with the tiling directions in the convolution kernel template (See Algorithm 3). Assuming the block sizes are NB, KB and CB, respectively, the three innermost loops compute the partial sum of a $NB * KB$ block and update the output with it. The outer loops control the traversal of different blocks.

2) Vectorization: Exploiting the capability of performing operations on multiple data elements simultaneously is important to improve the throughput of arithmetic operations and memory accesses. Modern CPU usually has vector instruction extensions, such as the Advanced Vector Extensions (AVX) [31] for Intel CPU and the NEON [32] for Arm CPU. The vector instructions can be generated by host compilers through embedding special functions named 'intrinsic' in code.

The innermost loop usually performs the same type operation on different elements that resides in memory contiguously, such as the multiplications between input values and weights in both convolution and linear transformation kernels, and the arithmetics in some element-wise kernels. It is suitable to utilize the vector instructions to boost the performance. Acorns will find loops with *vectorizing* direction and try to vectorize the inside expression by replacing it with corresponding intrinsics. In the code at Line 13 in Algorithm 3, the product between an input element and a weight is added to sum. Acorns will recognize this arithmetic operations and use the Fused Multiply-Add (FMA) intrinsic to replace it. Weights and inputs are also loaded through intrinsics before the computation. When combining vectorization and tiling, Acorns ensures the tiling size will be multiple of the width of used vector intrinsics exactly.

3) Auto-tuning: The tiling size has a great impact on the performance [28], [29] while it is not straightforward to determine the best tiling size for each level of the loop nests. In Acorns, we exploit the auto-tuning technique to search the best tiling size. The brute-force method tries every legal tiling configuration and picks out the one with the best performance. But it will take significant time to complete the searching as the search space is defined by the combination of all possible tiling sizes of each loop level. For real-world CNNs [24], [33], the number of legal candidates for one single convolution operator can easily reach several hundred.

We propose an empirical model to reduce the search space for Acorns. The core idea is to avoid trying candidate kernels that cannot achieve effective data reuse on specific platforms. Specifically, the model calculates the working set size of computing the $NB * KB$ partial sum block, which is (NB*KB+NB*CB+CB*KB)*sizeof(float), and checks if it exceeds the capacity of a specific level of data cache. The other constrain is the working set of the two innermost loops (loops of NB and KB) should fit in another higher level data cache, which is calculated as (NB*KB+NB+KB)*sizeof(float).

We choose the L2 and L1 data cache for the model and evaluate it on all 85 different convolution kernels from ResNet50 [33] and DenseNet-121 [24]. We find the best tiling sizes are always kept by the model and the search space can be reduced by up to 45%. On average, 30% candidates are skipped without being tested by Acorns.

4) Weight Packing: The access pattern of weight tensor changes after tiling. The weight tensor was accessed one by one in specific order (e.g., KRSC) with the values arranged in a compatible way. After tiling, weight tensor will be accessed block by block. With the original layout of weight tensor, this access pattern will result in high-stride accesses and potential TLB misses.

To resolve this problem, Acorns will adjust the layout of weight tensor automatically to ensure the contiguous accesses. The weight tensor will be packed into blocks with the same size as the related loop tiles, and the blocks will be stored one by one with respect to the access order. As the weight tensors are fixed during the inference phase, the packing procedure can be completed ahead of the actual inference.

5) Multithreading: Besides capturing the data-level parallelism through the vectorization, Acorns also exploits the loop-level parallelism exposed in kernels via multithreading. Acorns provides the *parallelization* direction to annotate loops that can be performed in parallel, and distributes the computation to multiple threads via OpenMP.

As the false sharing problem can cause significant performance degradation [34], we should avoid different threads writing memory locations within the same cacheline. Thus, the loops of C (K for mm and convolution) are excluded from parallelization, and we consider the loops of N. For element-wise kernels, annotating loops of N is safe as the size of channel dimension (usually between 16 to 1024) is large enough to avoid false sharing. For convolutions, as non-zero values residing in one $R * S$ block in the input tensor contribute to the same output spatial location, Acorns arranges each thread to compute the result of its own BN block and store it in a private buffer. A merging step will merge local results of different threads together and produce

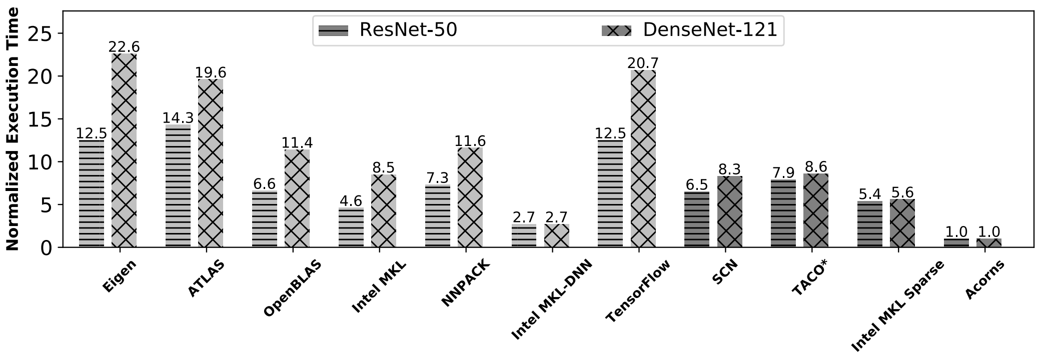


Figure 5. The single-thread performance of Acorns and baseline methods. The execution time of baseline methods is normalized to Acorns' time.

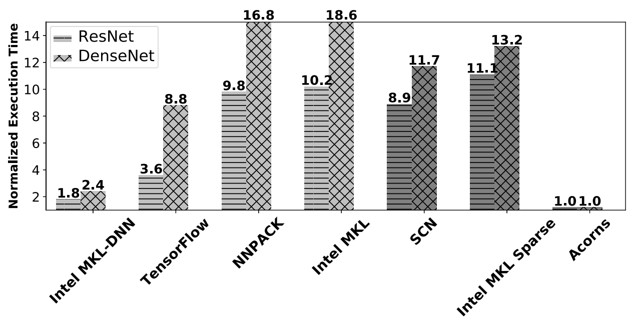


Figure 6. The multithreading performance of Acorns and baseline methods

the final outputs. The extra merging step brings some overheads and we compare the multithreading speedups with other baselines in Section V-B2.

V. Evaluations

To evaluate the techniques described in this paper, we use Acorns as well as several existing frameworks and libraries to execute convolutional neural networks with real-world sparse input data and compare their performance. We demonstrate in Section V-B Acorns can generate programs that outperform all baseline methods in performance with notable margins. We further demonstrate how much each part of Acorns contributes to the overall performance gain in Section V-C, and the overhead of data layout transformation in Section V-D. Finally, we show in Section V-E that our technique is able to exploit different degrees of sparsity and transform it to actual performance improvement effectively.

A. Methodology

We pick ten widely used and representative neural network frameworks and libraries as baselines to evaluate the performance improvement of Acorns. These baselines can be divided into two categories based on how they treat the sparsity. The sparsity-aware methods exploit the input sparsity in the computation. On the contrary, sparsity-unaware methods do not realize the sparsity and treat the input as dense data.

There are not many works utilizing input sparsity in deep learning workloads on CPU. SparseConvNet (SCN) [25] supports end-to-end sparse inference and utilizes the matrix-matrix multiplication kernel in PyTorch [13] to implement sparse convolution. Intel MKL provide heavily optimized sparse kernels (Intel MKL-Sparse) [35] for Intel processors and the SpMM kernel is used to implement mm and convolution operators. Since SpMM supports several different sparse matrix formats (CSR, CSC, COO), we pick out the one with the best performance to represent Intel MKL-Sparse. TACO [36] is a recently developed compiler that is capable of generating optimized kernels for tensor expressions with mixes of dense and sparse tensors.

As for sparsity-unaware methods, we choose six libraries, including Intel MKL-DNN [15], NNPACK [16], Eigen [37], Intel MKL [38], OpenBLAS [39] and ATLAS [40], and two frameworks, Caffe [12] and Google's TensorFlow [11]. Specifically, Caffe is only used as a front-end to parse the testing neural networks and call the corresponding computing kernels in the baseline libraries for operators.

The experiments use two state-of-the-art CNNs, ResNet-50 [33] and DenseNet-121 [24], in the LiDAR-based detection task. The input data comes from the widely used KITTI dataset [23], and it contains 7518 images with spatial size $1400 * 700$. The average sparsity is 0.79, meaning an input image has only 21% valid regions on average. To generate sparse kernels, Acorns uses randomly generated images with the average sparsity (79%). For a more comprehensive understanding of Acorns, in Section V-E, we also use randomly generated sparse images with a wider spectrum of sparsity degrees than KITTI to analyze Acorns' performance.

We run all experiments on a four-socket, 32-core Intel Xeon E7-4809 v3 CPU machine with 32GB of main memory. The CPU supports Intel Advanced Vector Extensions 2 (AVX2) [41], a 256-bit width vector instruction extension.

Table I
THE MULTITHREADING SPEEDUP OF ACORNS AND BASELINES

Speedup	ResNet	DenseNet
Intel MKL-DNN	5.4	3.3
TensorFlow	12.3	7.1
NNPACK	2.6	2.1
Intel MKL	1.6	1.4
SCN	2.6	2.2
Intel MKL-Sparse	1.7	1.3
Acorns	3.6	3.0

Table II
THE OVERALL PERFORMANCE IMPROVEMENT BREAKDOWN. *indi* REFERS TO INDIVIDUAL SPEEDUP, AND *accu* REFERS TO ACCUMULATED SPEEDUP

Speedup	ResNet		DenseNet	
	indi	accu	indi	accu
op fusion	1.02	1.02	1.06	1.06
vectorization	2.16	2.21	1.34	1.42
loop tiling&auto-tuning	2.29	5.05	2.47	3.51
weight packing	1.20	6.06	1.19	4.17
multithreading	3.55	21.51	3.03	12.64

Each core has 32KB of L1 data cache and 256KB of L2 cache. A 20MB of L3 cache is shared among 8 cores.

B. Overall Performance

In this section, we compare the performance of programs generated by Acorns with baseline methods. We use the time of running the models on all 7518 images in the KITTI test dataset to evaluate the performance.

1) Single-Thread: Fig. 5 displays the speedups achieved by Acorns in the single-thread situation. The performance of seven sparsity-unaware methods is represented by light-grey bars, and sparsity-aware methods are represented by dark-grey bars in the right part. As TACO does not support spatial convolutions (R>1 or S>1), we show its normalized time of the left supported kernels. It is clear Acorns achieves the best performance among all methods with notable speedups.

Compared with sparsity-unaware methods, Acorns achieves 2.7 to 22.6× speedups, demonstrating its capability to exploit the sparsity and transform it to actual performance improvement. It is notable that Acorns achieves 2.7× speedup over Intel MKL-DNN [15]. Besides, Intel MKL also shows performance comparable to sparsity-aware methods, demonstrating the difficulty for sparse computation in surpassing these optimized and carefully tuned methods.

Compared with the three sparsity-aware baselines, Acorns achieves 5.4 to 8.3× performance improvement, verifying the new data layout and collaborative template-based kernel generation can serve as a better solution to sparse inference. Among them, TACO is incapable of generating all kernels of operators in the testing neural networks. We also analyze the generated kernel code and find TACO lacks several performance-critical optimizations, including tiling and vectorization, which contribute a lot to the performance gain (See Section V-C).

Acorns shows 5.4 to 5.6× speedups over Intel MKL-Sparse. Among supported sparse formats, COO performs best in evaluations. Our format reduces memory traffic as non-zero locations of different channels are shared. Equipped with tuning-assisted loop tiling and explicit vectorization, Acorns can realize better utilization of registers and cache. We conduct cache-related profiling to analyze the memory access efficiency and the results are displayed in Fig. 7. Acorns show significantly less L1 and last-level cache references, and the reduced miss on L1 data cache and LLC demonstrates enhanced data locality.

The programs generated by Acorns runs 6.5 to 8.3× faster than SCN [25]. We find the building procedure to construct the hashtable-based representations incurs non-trivial overhead and takes 28% and 33% time in the ResNet-50 and DenseNet-121 inference, respectively. Moreover, the performance also suffers from SCN's insufficiently optimized kernels. It calls the matrix multiplication kernel $R*S$ times to complete convolution, wasting data reuse cross convolution kernels and resulting in worse performance. Acorns achieves an average 5.4× speedup on all 171 convolution kernels from ResNet-50 and DenseNet-121.

2) Multithreading: We start from 2 threads and double the thread number until it reaches 32. The best performance of each method in this process is recorded as its multithreading performance. Fig. 6 shows the multithreading speedups of Acorns over baselines. Eigen is skipped as its kernels used in the experiment do not support multithreading. TACO is also skipped for its incomplete support for all required kernels. Among dense BLAS libraries, we only keep the best-performing Intel MKL as the representative.

Acorns achieves the best performance on all models again, showing 1.8 to 18.6× speedups. Table I displays the speedups over its single-thread performance for each method. Except TensorFlow and NNPACK which implement own thread pool for better scalability, other methods utilize OpenMP for parallelization. While the extra merging step incurs some overheads, Acorns achieves the best multithreading speedup among sparsity-aware methods.

C. Performance Contribution Analysis

In this section, we break down the overall performance gain achieved by Acorns and analyze how much each component contributes to it. The optimizations are enabled one by one, and Acorns generates a program at each step. We

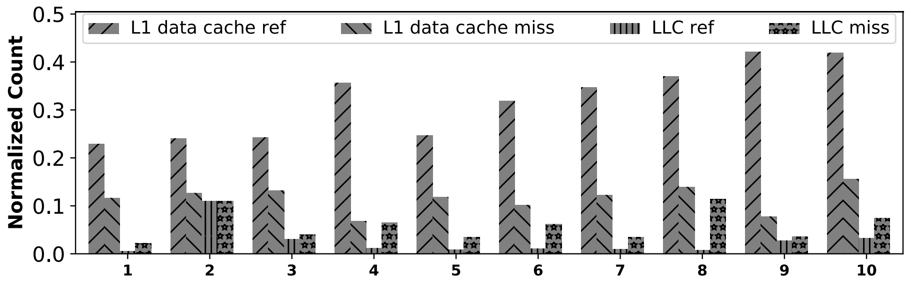


Figure 7. The cache profiling results of ten convolution kernels (along x axis) from ResNet-50 and DenseNet-121. We normalize Acorns' values to that of Intel MKL-Sparse, thus bars lower than 1 mean Acorns' values are smaller.

evaluate these generated programs on the KITTI dataset and the results are shown in Table II. From top to bottom, each line shows the performance improvement of an additional optimization. Both the individual speedup (speedup over the prior one) and the accumulated speedup (speedup over the baseline) are listed.

The benefit of operator fusion depends on the neural network architecture and the fused operators take 0.6% and 1% of computations in ResNet-50 and DenseNet-121, respectively. Although the improvements are small, we argue the fused operators would make a larger performance impact with more optimizations enabled, since these operators are insensitive to the following optimizations due to their low computing intensity and little data reuse. The vectorization and loop tiling with auto-tuning contribute most of the performance improvement except multithreading. It is worth noting that although the optimization techniques themselves are not original, we show building an end-to-end framework where they can work collaboratively is quite effective to solve the problem of sparse inference of deep neural networks.

D. Layout Transformation Overhead

In experiments, the transformation operators are inserted after the pooling and spatial convolutions, and the reverse transformation operators are inserted before pooling. All inserted transformation operators take 7.9% and 20.0% overall time in the inference of ResNet-50 and DenseNet-121, respectively. The reverse-transformation operators are more expensive duo to the irregular writing pattern and contribute more than 50% of transformation overhead. Since there are much more poolings in DenseNet than ResNet (4 vs 1), transformations in DenseNet incurs more overhead.

E. Performance with Varying Sparsity

In this section, we investigate Acorns' capability in exploiting different degrees of sparsity. As images in KITTI only cover a narrow range of sparsity (0.57-0.93), we synthesis images with sparsity ranging from 0.1 to 0.9 as input. Since different sparsity levels impacts the tuning results, Acorns runs the tuning procedure and generates a program

Table III
THE SPEEDUP ON DIFFERENT LEVELS OF SPARSITY

Sparsity	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
MKL-DNN	0.9	1.0	1.1	1.3	1.4	1.9	2.3	3.3	5.3
MKL-Sparse	8.0	8.3	8.1	8.0	7.4	7.7	7.2	6.8	5.7

for each sparsity level. We run the generated programs of ResNet-50 over 1000 images and compute the average time to evaluate the performance. Intel MKL-DNN and MKL-Sparse are chosen as baselines.

Table III displays the experiment result. Acorns can match Intel MKL-DNN when sparsity reaches 0.2 and achieve speedups under higher sparsity, demonstrating the optimizing capability of Acorns is not only effective for some specific levels of sparsity. The speedups over Intel MKL-Sparse verify the advantages of Acorns in generating and tuning sparse kernels for specific levels of sparsity. We also evaluate the performance loss when the sparsity level to which Acorns tunes mismatches the sparsity level at runtime. On average this causes about 7% performance degradation. The most performance loss is 21% and happens when using kernels tuned to 0.1 sparsity for inputs with 0.9 sparsity.

VI. RELATED-WORK

A. Efforts to exploit sparsity in neural networks

Efforts towards exploiting the sparsity in neural networks are most similar to Acorns. spg-CNN [42] exploits the sparsity produced in the back-propagation phase of training and reforms the gradient computation to a composition of small dense matrix multiplications. Optimizations including data layout transformation, vectorization and tiling are leveraged. Acorns targets the sparsity in input data and focuses on the inference phase. Upon the designed sparse data layout, Acorns proposes a template-based generating method to produce efficient kernels in neural networks. SCN [25] designs a hashtable-based sparse tensor representation and utilizes the computing kernels in PyTorch [13] to implement sparse operators. We demonstrate its inefficiency by showing its incapability to improve the performance compared to highly tuned dense library [15]. SBNet [8] divides tensors with block-structured sparsity into dense grids and uses the optimized library [43] for dense computation. Acorns do not only target specific spatial sparsity and is able to generate sparse kernels with better performance that tuned dense ones. Shi.et al [44] propose to skip zeros in operators' inputs on-the-fly and utilize SIMD to implement sparse kernels. But the insufficient designs in optimization result in limited speedups. TACO [36], [45] is a compiler to generate optimized kernels for tensor expressions. Acorns targets sparse inference of neural networks and is distinguished by the new designed data layout and the template-based optimizing kernel generator.

There are other efforts devoted to exploiting the sparsity in weight tensors. Park et al. [46] and Chen et al. [47] develop sparse convolutions with data placement arrangement [47] and tiling [46], [47] based on the CSR format for CPU and GPU, respectively. Acorns utilizes the domain-specific sparsity structure and organize sparse data in a new data layout. Liu et al [48] propose to customize sparse kernels by removing redundant instructions according to the specific sparse weights. Acorns exploits the sparsity in input data, which varies with different inputs and is out of the capability of that method.

B. Efforts to optimize sparse computation

Many sparse matrix formats have been proposed to support the representation and computation of sparse matrices. Compressed sparse row (CSR), compressed sparse column (CSC), and coordinate (COO) are most popular formats and are supported by most of computation libraries [35], [49]. Several efforts seek to optimize specific sparse operations by designing new formats, such as ELLPACK [50] and compressed sparse block (CSB) [51] for sparse matrix-vector multiplication (SpMV), and skyline [52] for Cholesky decomposition and LU decomposition. Our data layout for sparse tensors exploits the domain-specific 'channel consistency' in sparse inputs of neural networks and works collaboratively with designed optimization sequence to boost the performance of sparse kernels. Many efforts have been directed to implement efficient sparse operations with optimizing transformations, such as SpMV [50], [53]–[57] and SpMM [58]–[62]. Optimizing techniques like tiling [58] and auto-tuning [57] are also used to enhance the performance. Compared with these efforts, Acorns focuses on generating efficient sparse kernels used in neural network inference, and the generated kernels show notable performance improvement over existing sparse library [35] and compiler [36].

C. Deep Learning Compilers and Frameworks

As far as we know, no existing frameworks or compilers provide sufficient support for sparse input data in deep learning workloads, and Acorns is the first compiler to exploit the input sparsity to accelerate the inference. Several popular frameworks [11]–[14], [20], [63] allow users to construct neural networks using their APIs and delegate the computation to vendor libraries [15], [16], [38], [39], [43]. TensorFlow [11] provides a coo-based sparse tensor representation, but most of common operators do not support it, making end-to-end sparse inference impossible. Compared with them, Acorns exploits the input sparsity to accelerate the inference and is able to generate sparse kernels that have better performance than existing libraries.

Some works are devoted to generating efficient neural network kernels. TVM [64], [65] provides a DSL for users to express the computation and the transformations. It can generate inference programs by using graph-level transformations and tuning kernels for specific platforms. Tensor Comprehension [26] transforms tensor expressions into a polyhedral representation and the kernel code is generated by a jit compiler. Other works [66]–[68] design multi-level intermediate representations (IR) and generate LLVM IR [69] to utilize the low-level optimizations and portability of LLVM. Acorns shares some optimizing transformations with them, like loop tiling, multithreading parallelization, and data packing. But Acorns focuses on the sparsity in input data and achieves notable performance improvement by designing the data layout for sparse tensors and the optimizing sequence to exploit the potential performance gain effectively.

VII. Conclusion and Future Work

In this paper, we propose a framework to accelerate the neural network inference by exploiting the sparsity in input data. The designed data layout for sparse tensors exploits the domain-specific sparsity structure and enables performance-critical optimizations. Acorns performs optimizing transformations on kernel templates and utilizes auto-tuning to generate efficient inference code for neural networks. Comprehensive evaluations demonstrate the generated programs achieve significant performance improvement over state-of-the-art methods.

In the future, we plan to enhance Acorns from several perspectives, such as an unified kernel representation and new code generators for different platforms like GPU.

Acknowledgment

We thank the reviewers for the valuable feedback. This work is supported by National Key R&D Program of China under Grant No.2017YFB1003103, Science Fund for Creative Research Groups of the National Natural Science Foundation of China under Grant No.61521092 and the Key Program of National Natural Science Foundation of China under Grant No.61432018 and No.61432016.

References

[1] Y. Taigman, M. Yang, M. Ranzato, and L. Wolf, "Closing the gap to human-level performance in face verification. deepface," in *IEEE Computer Vision and Pattern Recognition (CVPR)*, 2014.

[2] G. Song, Y. Liu, M. Jiang, Y. Wang, J. Yan, and B. Leng, "Beyond trade-off: Accelerate fcn-based face detector with higher accuracy," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 7756–7764.

[3] C. C. Loy, T. Xiang, and S. Gong, "Multi-camera activity correlation analysis," in *2009 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2009), 20-25 June 2009, Miami, Florida, USA*, 2009, pp. 1988–1995. [Online]. Available: https://doi.org/10.1109/CVPRW.2009.5206827

[4] A. Conneau, H. Schwenk, L. Barrault, and Y. Lecun, "Very deep convolutional networks for text classification," in *European Chapter of the Association for Computational Linguistics EACL'17*, 2017.

[5] X. Chen, H. Ma, J. Wan, B. Li, and T. Xia, "Multi-view 3d object detection network for autonomous driving," in *IEEE CVPR*, vol. 1, no. 2, 2017, p. 3.

[6] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[7] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro *et al.*, "Applied machine learning at facebook: a datacenter infrastructure perspective," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 620–629.

[8] M. Ren, A. Pokrovsky, B. Yang, and R. Urtasun, "Sbnet: Sparse blocks network for fast inference," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[9] B. Graham, "Spatially-sparse convolutional neural networks," *arXiv preprint arXiv:1409.6070*, 2014.

[10] T. Dekel, C. Gan, D. Krishnan, C. Liu, and W. T. Freeman, "Sparse, smart contours to represent and edit images," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 3511–3520.

[11] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: a system for large-scale machine learning." in *OSDI*, vol. 16, 2016, pp. 265–283.

[12] 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 22nd ACM international conference on Multimedia*. ACM, 2014, pp. 675–678.

[13] Facebook, "Pytorch," 2018. [Online]. Available: https://pytorch.org/

[14] ——, "Caffe2," 2018. [Online]. Available: https://caffe2.ai/

[15] E. Georganas, S. Avancha, K. Banerjee, D. Kalamkar, G. Henry, H. Pabst, and A. Heinecke, "Anatomy of high-performance deep learning convolutions on simd architectures," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. IEEE Press, 2018, p. 66.

[16] M. Dukhan, "Nnpack," 2018. [Online]. Available: https://github.com/Maratyszcza/NNPACK/

[17] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[18] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *International Conference on Machine Learning*, 2015, pp. 448–456.

[19] W. F. Tinney and J. W. Walker, "Direct solutions of sparse network equations by optimally ordered triangular factorization," *proc. IEEE*, vol. 55, no. 11, pp. 1801–1809, 1967.

[20] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *arXiv preprint arXiv:1512.01274*, 2015.

[21] A. Heinecke, G. Henry, M. Hutchinson, and H. Pabst, "Libxsmm: accelerating small matrix multiplications by runtime code generation," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 2016, p. 84.

[22] Waymo, "Recreating the self-driving experience: the making of the waymo 360 video," 2018. [Online]. Available: https://medium.com/waymo/recreating-the-self-driving-experience-the-making-of-the-waymo-360-video-37a80466af49

[23] A. Geiger, P. Lenz, and R. Urtasun, "Are we ready for autonomous driving? the kitti vision benchmark suite," in *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*. IEEE, 2012, pp. 3354–3361.

[24] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks." in *CVPR*, vol. 1, no. 2, 2017, p. 3.

[25] B. Graham and L. van der Maaten, "Submanifold sparse convolutional networks," *arXiv preprint arXiv:1706.01307*, 2017.

[26] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen, "Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions," *arXiv preprint arXiv:1802.04730*, 2018.

[27] M. Wolfe, "More iteration space tiling," in *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*. ACM, 1989, pp. 655–664.

[28] J. J. Navarro, T. Juan, and T. Lang, "Mob forms: a class of multilevel block algorithms for dense linear algebra operations," in *Proceedings of the 8th international conference on Supercomputing*. ACM, 1994, pp. 354–363.

[29] M. D. Lam, E. E. Rothberg, and M. E. Wolf, "The cache performance and optimizations of blocked algorithms," in *ACM SIGARCH Computer Architecture News*, vol. 19, no. 2. ACM, 1991, pp. 63–74.

[30] S. Coleman and K. S. McKinley, "Tile size selection using cache organization and data layout," in *ACM SIGPLAN Notices*, vol. 30, no. 6. ACM, 1995, pp. 279–290.

[31] Intel, "Intel-avx," 2019. [Online]. Available: https://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions

[32] ARM, "Arm-neon," 2019. [Online]. Available: https://developer.arm.com/architectures/instruction-sets/simd-isas/neon

[33] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[34] M. Scott and W. Bolosky, "False sharing and its effect on shared memory performance," in *Proceedings of the USENIX Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS)*, vol. 57, 1993, p. 41.

[35] Intel, "Intel-mkl sparse kernels," 2018. [Online]. Available: https://software.intel.com/en-us/mkl-developer-reference-c-sparse-blas-level-2-and-level-3-routines

[36] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe, "The tensor algebra compiler," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, p. 77, 2017.

[37] Eigenteam, "Eigen," 2018. [Online]. Available: http://eigen.tuxfamily.org/index.php

[38] Intel, "Intel-mkl," 2018. [Online]. Available: https://software.intel.com/en-us/mkl-developer-reference-c-blas-level-3-routines

[39] X. Zhang, "Openblas," 2018. [Online]. Available: https://github.com/xianyi/OpenBLAS/

[40] R. C. Whaley and A. Petitet, "Minimizing development and maintenance costs in supporting persistently optimized BLAS," *Software: Practice and Experience*, vol. 35, no. 2, pp. 101–121, February 2005.

[41] Intel, "Intel-avx2," 2019. [Online]. Available: https://software.intel.com/en-us/cpp-compiler-developer-guide-and-reference-intrinsics-for-intel-advanced-vector-extensions-2

[42] S. Rajbhandari, Y. He, O. Ruwase, M. Carbin, and T. M. Chilimbi, "Optimizing cnns on multicores for scalability, performance and goodput," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017*, 2017, pp. 267–280. [Online]. Available: https://doi.org/10.1145/3037697.3037745

[43] 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.

[44] S. Shi and X. Chu, "Speeding up convolutional neural networks by exploiting the sparsity of rectifier units," *arXiv preprint arXiv:1704.07724*, 2017.

[45] F. Kjolstad, P. Ahrens, S. Kamil, and S. Amarasinghe, "Sparse tensor algebra optimizations with workspaces," *arXiv preprint arXiv:1802.10574*, 2018.

[46] J. Park, S. Li, W. Wen, P. T. P. Tang, H. Li, Y. Chen, and P. Dubey, "Faster cnns with direct sparse convolutions and guided pruning," *arXiv preprint arXiv:1608.01409*, 2016.

[47] X. Chen, "Escort: Efficient sparse convolutional neural networks on gpus," *arXiv preprint arXiv:1802.10280*, 2018.

[48] 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, pp. 806–814.

[49] Nvidia, "cusparse," 2018. [Online]. Available: https://docs.nvidia.com/cuda/cusparse/index.html

[50] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *Proceedings of the conference on high performance computing networking, storage and analysis*. ACM, 2009, p. 18.

[51] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, "Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks," in *SPAA 2009: Proceedings of the 21st Annual ACM Symposium on Parallelism in Algorithms and Architectures, Calgary, Alberta, Canada, August 11-13, 2009*, 2009, pp. 233–244. [Online]. Available: https://doi.org/10.1145/1583991.1584053

[52] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. V. der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. Philadelphia, PA: SIAM, 1994.

[53] J. L. Greathouse and M. Daga, "Efficient sparse matrix-vector multiplication on gpus using the csr storage format," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 2014, pp. 769–780.

[54] W. Liu and B. Vinter, "Csr5: An efficient storage format for cross-platform sparse matrix-vector multiplication," in *Proceedings of the 29th ACM on International Conference on Supercomputing*. ACM, 2015, pp. 339–350.

[55] D. Merrill and M. Garland, "Merge-based sparse matrix-vector multiplication (spmv) using the csr storage format," in *ACM SIGPLAN Notices*, vol. 51, no. 8. ACM, 2016, p. 43.

[56] B. Xie, J. Zhan, X. Liu, W. Gao, Z. Jia, X. He, and L. Zhang, "Cvr: Efficient vectorization of spmv on x86 processors," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. ACM, 2018, pp. 149–162.

[57] H. Yoshizawa and D. Takahashi, "Automatic tuning of sparse matrix-vector multiplication for CRS format on gpus," in *15th IEEE International Conference on Computational Science and Engineering, CSE 2012, Paphos, Cyprus, December 5-7, 2012*, 2012, pp. 130–136. [Online]. Available: https://doi.org/10.1109/ICCSE.2012.28

[58] H. M. Aktulga, A. Buluç, S. Williams, and C. Yang, "Optimizing sparse matrix-multiple vectors multiplication for nuclear configuration interaction calculations," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE, 2014, pp. 1213–1222.

[59] C. Hong, A. Sukumaran-Rajam, I. Nisa, K. Singh, and P. Sadayappan, "Adaptive sparse tiling for sparse matrix multiplication," in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. ACM, 2019, pp. 300–314.

[60] C. Yang, A. Buluç, and J. D. Owens, "Design principles for sparse matrix multiplication on the GPU," in *Euro-Par 2018: Parallel Processing - 24th International Conference on Parallel and Distributed Computing, Turin, Italy, August 27-31, 2018, Proceedings*, 2018, pp. 672–687. [Online]. Available: https://doi.org/10.1007/978-3-319-96983-1_48

[61] G. O. López, F. Vázquez, I. García, and E. M. Garzón, "Fastspmm: An efficient library for sparse matrix matrix product on gpus," *Comput. J.*, vol. 57, no. 7, pp. 968–979, 2014. [Online]. Available: https://doi.org/10.1093/comjnl/bxt038

[62] C. Hong, A. Sukumaran-Rajam, B. Bandyopadhyay, J. Kim, S. E. Kurt, I. Nisa, S. Sabhlok, Ü. V. Çatalyürek, S. Parthasarathy, and P. Sadayappan, "Efficient sparse-matrix multi-vector product on gpus," in *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing, HPDC 2018, Tempe, AZ, USA, June 11-15, 2018*, 2018, pp. 66–79. [Online]. Available: https://doi.org/10.1145/3208040.3208062

[63] L. Truong, R. Barik, E. Totoni, H. Liu, C. Markley, A. Fox, and T. Shpeisman, "Latte: a language, compiler, and runtime for elegant and efficient deep neural networks," *ACM SIGPLAN Notices*, vol. 51, no. 6, pp. 209–223, 2016.

[64] T. Chen, T. Moreau, Z. Jiang, H. Shen, E. Yan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "Tvm: end-to-end compilation stack for deep learning," in *SysML Conference*, 2018.

[65] Y. Liu, Y. Wang, R. Yu, M. Li, V. Sharma, and Y. Wang, "Optimizing CNN model inference on cpus," *CoRR*, vol. abs/1809.02697, 2018. [Online]. Available: http://arxiv.org/abs/1809.02697

[66] Google, "Xla: Domain-specific compiler for linear algebra to optimizes tensorflow computations." 2018. [Online]. Available: https://www.tensorflow.org/performance/xla/

[67] R. Wei, V. Adve, and L. Schwartz, "Dlvm: A modern compiler infrastructure for deep learning," *arXiv preprint arXiv:1711.03016*, 2017.

[68] N. Rotem, J. Fix, S. Abdulrasool, S. Deng, R. Dzhabarov, J. Hegeman, R. Levenstein, B. Maher, S. Nadathur, J. Olesen *et al.*, "Glow: Graph lowering compiler techniques for neural networks," *arXiv preprint arXiv:1805.00907*, 2018.

[69] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society, 2004, p. 75.