Fast Sparse ConvNets

Erich Elsen*
DeepMind
London, UK
eriche@google.com

Marat Dukhan*
Google
Mountain View, CA
maratek@google.com

Trevor Gale*
Google
Mountain View, CA
tgale@google.com

Karen Simonyan
DeepMind
London, UK
simonyan@google.com

Abstract

Historically, the pursuit of efficient inference has been one of the driving forces behind the research into new deep learning architectures and building blocks. Some of the recent examples include: the squeeze-and-excitation module [16], depthwise separable convolutions in Xception [4], and the inverted bottleneck in MobileNet v2 [35]. Notably, in all of these cases, the resulting building blocks enabled not only higher efficiency, but also higher accuracy, and found wide adoption in the field. In this work, we further expand the arsenal of efficient building blocks for neural network architectures; but instead of combining standard primitives (such as convolution), we advocate for the replacement of these dense primitives with their sparse counterparts. While the idea of using sparsity to decrease the parameter count is not new [42], the conventional wisdom is that this reduction in theoretical FLOPs does not translate into real-world efficiency gains. We aim to correct this misconception by introducing a family of efficient sparse kernels for several hardware platforms, which we open-source as part of the XNNPACK library for the benefit of the community. Equipped with our efficient implementation of sparse primitives, we show that sparse versions of MobileNet v1, MobileNet v2 and EfficientNet architectures substantially outperform strong dense baselines on the efficiency-accuracy curve. On Snapdragon 835 our sparse networks outperform their dense equivalents by 1.3 – 2.4× – equivalent to approximately one entire generation of improvement. We hope that our findings will facilitate wider adoption of sparsity as a tool for creating efficient and accurate deep learning architectures.

1. Introduction

Convolutional neural networks (CNNs) have proven to be excellent at solving a diverse range of tasks [2]. Standard network architectures are used in classification, segmentation, object detection and generation tasks [31, 26, 47]. Given their wide utility, there has been significant effort to design efficient architectures that are capable of being run on mobile and other low power devices while still achieving high classification accuracy on benchmarks such as ImageNet [34]. For example, MobileNets [15, 35] employ the depthwise separable convolutions introduced in [36] to significantly reduce resource requirements over previous architectures. Inference time and computational complexity in these architectures are dominated by the 1×1 convolutions, which directly map to matrix-matrix multiplications.

Weight sparsity is generally known to lead [3] to theoretically smaller and more computationally efficient (in terms of number of floating-point operations) models, but it is often disregarded as a practical means of accelerating models because of the misconception that sparse operations cannot be fast enough to achieve actual speedups during inference. To address this misconception and firmly establish sparsity as a tool in the deep learning practitioner’s arsenal, we introduce fast kernels for Sparse Matrix-Dense Matrix Multiplication (SpMM) specifically targeted at the acceleration of sparse neural networks. The main distinction of our SpMM kernel from prior art [30, 45] is that we focus on a different point in the design space. While prior work focused on extremely sparse problems (typically >99%, found in scientific and graph problems), we target the sparsity range of 70-95%, more common when inducing weight sparsity in neural networks. As a result our kernels significantly outperform the kernels generated by the TACO compiler [22].

Using these kernels, we demonstrate the effectiveness of weight sparsity across three generations of MobileNet [15, 35, 40] architectures. Sparsity leads to an improvement of approximately one whole generation in each architecture, with sparse EfficientNets being significantly more efficient than all previous models. These models represent a new generation of efficient CNNs, which reduces inference times by 1.3 – 2.4×, parameter counts by over 2× and number of floating-point operations (FLOPs) by up to 3× relative to the previous generations.

*These authors contributed equally to this work
Figure 1: MobileNet v1 and v2 and EfficientNet models. Sparse models: blue, dense models: red. Sparse models include the cost of storing the location of non-zeros for sparse tensors as a bitmask converted back into parameter count. That is every 32 values in the bitmask contributes one “parameter”.

2. Related Work

Improvements in convolutional network architectures [23, 37, 12, 17], as measured by increased classification accuracy on benchmark tasks such as ImageNet [34], have generally been concomitant with increases in model parameter counts, FLOPs and memory requirements. Recently this evolution has led to networks found through neural architecture search [50, 33] which can achieve over 82% top-1 accuracy, but require nearly 25 GFLOPs for one inference.

Given these prohibitive inference costs, there have been many lines of work attempting to improve CNN efficiency, which is often defined as one of three metrics:

1. Inference speedup on real hardware
2. Theoretical speedup through FLOPs reduction
3. Model size reduction

These axes are neither parallel nor orthogonal. The effect of (3) and (2) on (1) in particular can be quite complicated and highly varied depending on the hardware in question.

The MobileNet family of architectures [15, 35] has focused on improving efficiency by taking advantage of the depthwise separable convolutions introduced in [36], which can be thought of as a hand-crafted sparsification of full convolutions with a predefined sparse topology, and which are responsible for the parameter efficiency of these architectures. MobileNet v1 (MBv1) used layers of $1 \times 1$ convolutions followed by depthwise convolutions. MobileNet v2 (MBv2) introduced the inverted residual block which consists of a $1 \times 1$ convolution expanding the channel count, a depthwise convolution on the expanded channel count, and then a $1 \times 1$ convolution reducing the parameter count. Across MobileNet architectures, the depthwise convolutions account for only a small fraction of the total FLOPs, parameters, and inference time of these models. In MBv1, they account for less than 2% of the total FLOPs and in MBv2 less than 3%.

A different line of work attempted to make more efficient CNNs by directly pruning the weights of full convolutional filters accompanied by the necessary inference kernels [32, 24]. [32] was not able to accelerate $1 \times 1$ convolutions, [24] did not attempt it. The latter also required generating a new set of kernels for each instance of a model, which is often impractical for deployment. Due to the difficulty of accelerating sparse computation, channel pruning approaches have been preferred [10, 5, 28, 27, 41, 13]. These approaches prune away entire filters leaving the final model dense, and function more as an architecture search over channel counts.

Full Neural Architecture Search has also been applied directly to architectures resembling MBv2 resulting in MobileNet v3 [39], FBNet [44], and EfficientNet [40].

Alternatively, factorizations of the $1 \times 1$ convolutions have been considered in ShuffleNet [46] and Learnable Butterfly Factorizations [6]. ShuffleNet factorizes the weight matrix into a product of a permutation matrix and block diagonal matrix. Butterfly Factorizations factorize the weight matrix into a sequence of permutation matrices and weight matrices with special structure that can represent many common $O(N \log N)$ transforms such as Fast Fourier Transforms.
Work in Text-to-Speech (TTS) [21] demonstrated that increasing sparsity and concomitant increase in state size in RNN models lead to increased model quality for a given non-zero parameter count. They additionally demonstrated fast block-sparse matrix-vector (SpMV) multiplication routines necessary for RNN inference.

3. Methods

To understand how to design the most efficient convolutional models, we investigate both how to construct and train sparse MBv1, MBv2 and EfficientNet models and also the performance of our SpMM kernels.

3.1. Sparsifying Networks

We train on the ImageNet [34] dataset with standard augmentation and report top-1 accuracies on the provided 50k example validation set. To make the networks sparse we use the gradual magnitude pruning technique of [48].

We do not prune the first full convolution at the beginning of all three networks. Its overall contribution to the parameter count, FLOP count, and runtime is small and does not warrant introducing a new sparse operation. Instead, we implement a dense convolutional kernel which takes as input the image in the standard HWC layout and outputs the CHW layout consumed by the sparse operations in the rest of the network. In HWC layout, the values for different channels corresponding to one spatial location are adjacent in memory. In CHW layout, the values of all the spatial locations for one channel are adjacent in memory.

We also do not prune the squeeze-excitation [16] blocks in EfficientNet as they contribute <1% of the total FLOPs to the dense model. The last fully-connected layer in all models also contributes insignificantly (<1%) to the total FLOP count, but does contribute a significant fraction (20-50%) of total parameters, especially after the rest of the model is pruned. As we are concerned with maximizing top-1 accuracy for a given runtime, we do not prune the final layer in MobileNet v1 and v2 as doing so leads to a small decrease in top-1 accuracy. Standard EfficientNets do not scale the number of filters in the last convolution by the width of the model, however we find that when introducing sparsity it is beneficial to do this; in all sparse EfficientNet models we double the units from 1280 to 2560. We also find that it is possible to make the fully-connected layer sparse without loss of accuracy in EfficientNet, so we do so.

3.2. Kernel Implementation

A diagram of the $1 \times 1$ convolution as a SpMM is seen in figure 2. Our scheme requires activation tensors be stored in CHW format, in contrast to dense mobile inference li-
There are three key insights enabling the high performance of our kernels:

1. While the weight matrix is sparse, the activation matrix is dense. This means that we can perform vector loads from the activation matrix and process multiple spatial locations simultaneously.

2. By processing the matrix in the right order we can keep values that will be randomly accessed in the L1 cache, from which random access is fast and constant time.

3. When the number of input channels is small enough, prefetching from the activations can further reduce cache misses.

Figure 3 shows the memory read and write patterns of a few steps of the kernel. The figure shows 8 elements being processed together for visualization but 16 is more natural for the actual implementation as it corresponds to one cache line. The outer loop is over columns and the inner loop is over rows; this allows each strip of 16 spatial locations in the activations to remain in the L1 cache until it is no longer needed. In figure 3 steps 1 and 2 prime the cache, while subsequent steps 3 and 4 load all right hand side values from the L1 cache.

In addition to the vectorization in the HW dimension, taking advantage of small amounts of structure in the weight matrix can offer significant performance boosts by increasing data reuse after values are loaded into registers. Constraining the sparsity pattern so that multiple output or input channels all share the same zero/non-zero pattern creates ‘blocks’ in the weight matrix (see figure 3 right). Blocks in the output channel dimension allow for more data reuse than blocks in the input channel dimension. Experiments (see figure 5) show that either choice has the same effect on accuracy, so we implement output channel blocking with sizes of 2 and 4. Our nomenclature for kernels is to give their spatial vectorization width followed by the output channel block size – 16x2 means 16 pixels and 2 output channels are processed in the inner loop.

We implement the ARM kernels in C with NEON intrinsics unlike current production libraries [19, 7, 20] which rely on expert-optimized assembly.

3.3. XNNPACK Integration

Our kernels are available as part of XNNPACK [29]. We plan to release a TF-lite delegate that can run sparse models trained with the model pruning library in TensorFlow [1]. This includes conversion from a dense representation to a Block Compressed Sparse Row (BCSR)-like representation suitable for inference. In addition to the high performance 1×1 convolutions, we also provide all supporting CHW kernels – depthwise convolutions, global average pooling and a 3×3 stride-2 dense convolution – necessary for running all three generations of models. While we provide high performance versions of these kernels, we do not detail them here. They are included in end-to-end measurements.

3.4. WebAssembly

In some settings it is useful to run inference (and even training) of deep neural networks directly in web browsers. This is supported now by multiple frameworks.
Table 1: Comparison of dense and sparse model sizes, flops, and inference speeds. All input image sizes are 224x224. Sparse MBv1 models are 90% sparse in every layer, Sparse MBv2 models are 85% sparse. In sparse MBv1 models, layer 12 uses a block size of 4. This is almost as efficient as the models in 4.3 and matches the top-1 scores of the dense models more closely. In sparse MBv2 models, layers 11 and onwards use a block size of 2. The finally fully connected layer in all models is dense. All times are in milliseconds. Dense times on ARM are measured using XNNP ACK with intrinsics (as the sparse kernels are written with intrinsics) and TensorFlow Lite. Web Assembly results are measured on an Intel W-2135, dense times use Intel’s webml-polyfill [18] library. Sparse parameter counts include the overhead of sparsity storage as a bitmask for each sparse layer. EN-b1 is 85% sparse and unstructured, final FC layer is sparse.

‡This model is 80% sparse in all layers (except final fully-connected) and uses a block size of 1 everywhere.

†This is the cache aware MBv2 architecture described in section 4.4. It uses a block size of 1 throughout.

We also tried WebDNN [14], but it does not support some operations necessary to run MobileNet and EfficientNet models.

4. Results

In the main text we mainly include results for MBv1 and MBv2 due to space limitations. EfficientNets generally follow the same trends as MBv2 models, plots for EfficientNet can be found in the supplementary material. First we reveal performance results for our SpMM kernels, then we show how the networks respond to sparsity and then finally we combine this information to find the models with the lowest inference time.

including WebDNN [14], Tensorflow.js [38] and Webml-polyfill [18]. Frameworks generally support using WebAssembly (Wasm) [11] to run on CPUs or WebGL to run on GPUs. In this work we target web assembly backends and show that sparse vision networks significantly outperform their dense counterparts in this setting. In this setting our kernel decomposition strategy is similar, however due to current lack of vectorization support in Wasm, they are converted to scalar instructions. Due to the smaller register file, we find that unrolling by 8 instead of 16 is optimal. These kernels are available as part of XNNSPACK on github.
4.1. ARM Kernel Performance

We use both Ruy [20], the current TensorFlow Lite ARM64 backend written largely in hand-coded assembly, and dense XNNPACK with intrinsics as the dense baselines. For a sparse baseline we use the kernel generated by the TACO compiler [22]. We present results by plotting the FLOPs achieved at each layer in the model, with increasing depth to the right in figure 4. For MBv1 we use a width multiplier of 1.4 and 90% sparse and for MBv2 we use a width multiplier of 1.4 and 85% sparse as these configurations approximately match the top-1 accuracy of the width 1 dense models. The kernel variants that process 16 spatial locations at a time (e.g. 16x1, etc.) are the highest performing and all reported numbers are from these kernel variants. TACO results should be compared with the 16x1 kernels.

The raw performance of the sparse kernels falls in the range of 40–90% of the dense kernels. And as they must do much less work, when taking the sparsity of the layer into account, the effective FLOPs are in the 2–7× range. In MBv1 performance falls significantly in the last two layers of the model when the number of channels (1024) causes the size of one “strip” of spatial locations to exceed the size of the L1 cache. In MBv2 the sawtooth pattern is caused by the alternating expand and contract operations. The performance is higher for the expand kernels due to greater data reuse of each “strip” that is brought into the L1 cache.

4.2. Model Performance

The hyper-parameters used to train MBv1 and MBv2 are listed in table 2, they were found with a grid search on dense models with a width multiplier of 1.0 to reproduce the original results, which used RMSProp [43], with SGD with momentum. The same hyper-parameters are used to train sparse models. This change allows us to match or exceed the reported accuracies with only 38,000 iterations of training.

<table>
<thead>
<tr>
<th></th>
<th>MBv1</th>
<th>MBv2</th>
</tr>
</thead>
<tbody>
<tr>
<td>learning rate</td>
<td>.35 × 16 = 5.6</td>
<td>.24 × 16 = 3.84</td>
</tr>
<tr>
<td>momentum</td>
<td>0.9</td>
<td>0.92</td>
</tr>
<tr>
<td>l2 coefficient</td>
<td>5e-5</td>
<td>4e-5</td>
</tr>
</tbody>
</table>

Table 2: Hyper-parameters for MBv1 and MBv2 training. Learning rates are specified in a reduced space and then multiplied by a factor of 16 due to the batch size (4096). The learning rate schedule is a linear ramp for the first 8 epochs to the maximum value followed by step-wise decay at epochs 40, 75 and 95 by a factor of ten.

The hyper-parameters used to train EfficientNet are largely unmodified from their code release, with the exception of extending training from 350 to 650 epochs and increasing the learning rate decay exponent to .985 from .97 so that the learning rate decays more slowly. These changes do not improve the dense baseline.

We induce sparsity in MBv1 and MBv2 by extending training by a factor (along with learning rate anchor points) of four, we find this increases the performance of the sparse models, but not the baselines. We start the sparsification process at iteration 7,000 × 4 = 28,000 and stop at 28,000 × 4 = 112,000 with a pruning frequency of 2,000. For EfficientNet we start at iteration 23,000 and end at iteration 105,000, also with a pruning frequency of 2,000. See [48] for the meaning of these hyper-parameters.

We train on the ImageNet [34] dataset with standard data augmentation. Top-1 accuracies are reported on the validation set with center single-crops.

To understand the effect of block size, we plot in figure 5 accuracy against flops for different block sizes. In these plots, every sparse tensor in the network uses the same output channel block size. The tradeoff for block sparsity only appears to involve how many elements are in each block, and not their configuration. For example, in MBv1, the 1×4, 4×1 and 2×2 curves all lie on top of one another. The loss in accuracy due to blocking seems to decrease slightly for larger width models.

To understand how the sparsity level affects the efficiency of the models, we train models at 70%, 80% and 90% unstructured sparsity which is constant throughout the model. The results are plotted in figure 6. MBv1 and MBv2 are more efficient the more sparse they become, confirming that the results of [21] hold not just for RNNs, but also for convolutional models as well.

In figure 1 we plot Top-1 accuracy vs. FLOPs for all three generations of sparse and dense models. MobileNet v1 is 90% sparse, the other models are 80% sparse. A sparse MBv1 exceeds MBv2 in terms of FLOP and parameter efficiency; a sparse MBv2 matches EfficientNet in terms of FLOP and parameter efficiency; and a sparse EfficientNet exceeds all other models in both categories.

4.3. Model Design for Block Size

To design the models with the best top-1 accuracy vs. inference time frontiers we make the following assumptions to reduce the search space:

1. We leave the models themselves unchanged.
2. We induce the same level of sparsity in all 1 × 1 convolutions.

Then we do a search at width multiplier 1.4 over N models when there are N residual blocks in a model. An x-axis location of n corresponds to a model in which the first n residual blocks are unstructured and the last $N - n$ residual blocks have an output channel block size of 4. We train
Figure 5: Effect of block size on top-1 accuracy. It only matters how many elements are in a block, the configuration is unimportant.

Figure 6: Effect of sparsity on top-1 accuracy. The sparser a model is, the fewer flops it requires to achieve a given Top-1 accuracy.

Each model, note its top-1 accuracy and then measure its inference time. From this we can calculate the ratio of inference time reduction relative to a fully unstructured model and top-1 lost, which are plotted in figure 7. We choose the model with the highest ratio and train models at all widths with this choice. This amounts to making layers 6 and deeper blocked in MBv1 models and layers 11 and deeper blocked in MBv2.

4.4. Cache Aware Model Design

The sawtooth pattern in figure 4 is due to the large number of channels during the contract phase causing the size of stripe to exceed the size of the L1 cache. One possible solution, common with dense kernels, would be to split the matrix into N pieces such that each piece only accesses a number of channels that fit within the cache. However, the unstructured nature of the sparsity makes this complicated – there might load balancing issues and it would necessitate repacking the matrix. Instead of introducing more complexity into the software, we examine a simple modification to the MBv2 architecture to make it cache aware.

The inverted residual block introduced by MBv2 expands the number of channels before the depthwise convolution and then reduces them afterwards. The expansion factor is fixed at 6 for all layers. Once there are more than 512 channels, the size of the 32Kb L1 cache is exceeded. Additionally, once there are more than 256 channels, the data exceeds half of the cache precluding effective use of...
pre-fetching. To design an architecture that is aware of these constraints we take MBv2 and increase the expansion factor of early layers while reducing the expansion factor as the number of channels increases so that the number of expanded channels still fits within approximately half of the cache size. To compensate for the decrease in capacity this would otherwise cause, we also increase the depth of the model by adding one more layer with 32 channels, two more with 64 channels and three more each with 96 and 160 channels. The final architecture is in table 3.

Table 3: Architecture of cache aware MBv2 optimized for our sparse kernels. Each block is repeated n times and contains c channels that are expanded by a factor of e. The initial convolution in each block has stride s, all others have stride 1.

<table>
<thead>
<tr>
<th>Input</th>
<th>Operation</th>
<th>e</th>
<th>c</th>
<th>n</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>$224^2 \times 3$</td>
<td>conv2d</td>
<td>-</td>
<td>16</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>$112^2 \times 16$</td>
<td>Bottleneck</td>
<td>1</td>
<td>16</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$112^2 \times 16$</td>
<td>Bottleneck</td>
<td>8</td>
<td>24</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>$56^2 \times 24$</td>
<td>Bottleneck</td>
<td>8</td>
<td>32</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>$28^2 \times 32$</td>
<td>Bottleneck</td>
<td>4</td>
<td>64</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>$14^2 \times 64$</td>
<td>Bottleneck</td>
<td>3</td>
<td>96</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>$7^2 \times 96$</td>
<td>Bottleneck</td>
<td>2</td>
<td>160</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>$7^2 \times 160$</td>
<td>Bottleneck</td>
<td>2</td>
<td>320</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$7^2 \times 320$</td>
<td>conv2d 1x1</td>
<td>-</td>
<td>1280</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$7^2 \times 1280$</td>
<td>gavgpool</td>
<td>-</td>
<td></td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>$1^2 \times 1280$</td>
<td>conv2d 1x1</td>
<td>-</td>
<td>k</td>
<td>1</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 1 contains the timings for running our sparse models on a single big core of two different processors, a Snapdragon 835 and a Snapdragon 670. We compare them with MBv1 and MBv2 models from their official repositories [8, 9] run on the dense-inference TF Lite framework with the standard Ruy backend and also dense XNNPACK with intrinsics. Model files for all models in table 1 are available here.

Surprisingly, in the presence of sparsity MBV1 is approximately as efficient as MBV2 suggesting that a full Neural Architecture Search [49, 25] will likely lead to even more efficient models, which we leave to future work.

5. Conclusion

We demonstrate that for a constant computational budget, sparse convolutional networks are more accurate than dense ones; this corroborates the findings of [21], which demonstrated that for a set number of floating-point operations, sparse RNNs are more accurate than dense RNNs. We enable the use of weight sparsity to accelerate state-of-the-art convolutional networks by providing fast SpMM kernels along with all necessary supporting kernels for ARM processors. Combining quantization and sparsity is an interesting future direction, the smaller cache footprint afforded by quantization synergizes nicely with the sparse kernels. On Snapdragon 835 the sparse networks we present in this paper outperform their dense equivalents by $1.3 - 2.4 \times$ equivalent to approximately one entire generation of improvement. By overturning the misconception that “sparsity is slow”, we hope to open new avenues of research that would previously not be considered.
References


[8] Google, 2018. 8


[18] Intel. Webml-polyfill, 2019. 5


14638