B. Juurlink, W.Karl (Hrsg.)
Proceedings 28th PARS Workshop

Weight Pruning for Deep Neural Networks on GPUs

Thomas Hartenstein, Daniel Maier, Biagio Cosenza, Ben Juurlink
Embedded Systems Architecture, Technische Universität Berlin, Germany
thomas.hartenstein@campus.tu-berlin.de, {daniel.maier,cosenza,b.juurlink}@tu-berlin.de

Abstract: Neural networks are getting more complex than ever before, leading to resource-demanding training processes that have been the target of optimization. With embedded real-time applications such as traffic identification in self-driving cars relying on neural networks, the inference latency is becoming more important. The size of the model has been identified as an important target of optimization, as smaller networks also require less computations for inference. A way to shrink a network in size is to remove small weights: weight pruning. This technique has been exploited in a number of ways and has shown to be able to significantly lower the number of weights, while maintaining a very close accuracy compared to the original network. However, current pruning techniques require the removal of up to 90% of the weights, requiring high amount of redundancy in the original network, to be able to speedup the inference as sparse data structures induce overhead. We propose a novel technique for the selection of the weights to be pruned. Our technique is specifically designed to take the architecture of GPUs into account. By selecting the weights to be removed in adjacent groups that are aligned to the memory architecture, we are able to fully exploit the memory bandwidth. Our results show that with the same amount of weights removed, our technique is able to speedup a neural network by a factor of 1.57× given a pruning rate of 90% while maintaining the same accuracy when compared to state-of-the-art pruning techniques.

Keywords: deep neural network; pruning; GPUs; optimization

1 Introduction

Contemporary AI applications are often build using deep neural networks (DNNs). DNNs have improved over the last years becoming state-of-the-art not only for the majority computer vision algorithms but also they have been shown to give superior results in numerous other applications like speech recognition, natural language processing or the discovery of new drugs. In many of these applications, hard real-time deadlines have to be met in order to ensure user satisfaction or even prevent disastrous outcomes, e.g., for self-driving cars. However, to achieve better accuracy, the networks have become also more complex and the network sizes have grown significantly: While the AlexNet Caffemodel is over 200 MB in size, the improved VGG-16 Caffeemodel has already grown to more than 500 MB [HMD15]. More complex networks are composed of more layers and layers have become bigger, leading to resource-demanding training processes that have been the target of optimization in the past. However, for embedded real-time applications (e.g., traffic identification and object detection in self-driving cars) relying on neural networks, the

inference latency is more important. The size of a model has been identified as an important target of optimization as the size is directly related to the number of operations necessary for inference.

Research has shown that models contain a considerable amount of redundancy [De13]. Many connections in the neural network that represent the weights have no or only a minor role when deriving the result. These weights can be removed without affecting the accuracy of network significantly [HMD15; LDS90].

The optimization process of removing weights from a neural network is called *weight pruning*. The general concept of weight pruning is shown in Figure 1. All weights below a certain threshold, in this example 0.3, are removed from the network. Using this approach we can learn the important connections in the network. The result is a new network that contains only the relevant connections of the original network while connections with a negligible influence have been removed. Weight pruning is able to improve the memory usage, as less weights need to be stored in memory. Furthermore, the number of operations needed to compute the result of the network is reduced. The number of weights directly translates to the memory usage and is also closely related to the number of operations needed.

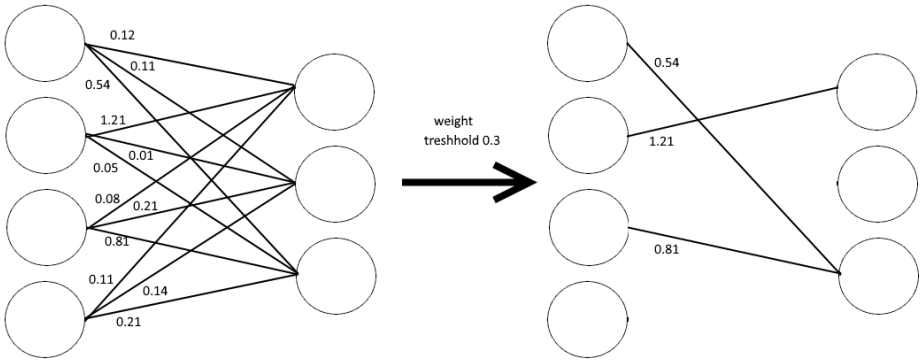

Fig. 1: Weight pruning removes weights below a certain threshold from a neural network.

We propose to use a new technique for weight pruning that overcomes limitations of the state-of-the-art pruning for GPUs [Yu17]. Memory-aware weight pruning is able to accelerate the inference time of deep neural networks by removing weights in continuous groups of multiple weights. These groups are optimized to match the memory architecture of GPUs.

In particular, we make the following contributions: 1. a novel weight pruning technique for neural networks on GPUs; and 2.an evaluation of our technique in terms of inference time and accuracy of the network.

This paper is organized as follows: In Section 2 the related work is introduced and we relate our work to the state-of-the-art. Section 3 introduces our technique for pruning of weights. We describe the experimental setup in Section 4 and show the results in Section 5. Finally, we conclude our work in Section 6.

2 Related Work

Neural networks contain a significant amount of redundant information. Therefore, the computational and memory requirements can both be optimized without a loss in accuracy [De13].

Redundancy in neural networks can be reduced using different techniques. One approach is quantization. By using less bits to store the weights of the network, the overall storage requirements are lowered. Gupta et al. use 16 bit fixed-point number representation for for the calculations of their neural network. [Gu15]. Gong et al. [Go18] show that a pre-trained neural network can be quantized to 8-bit without the necessity of having to retrain the network.

Another popular technique is pruning. Pruning is the removal of filters, weights or whole neurons from a network. The conceptual idea of removing weights is quite old [LDS90]. Pruning can be implemented in a variety of ways: One method is to remove complete filters from Convolutional Neural Networks [Hu18; Li17; Mo17]. Learning the important connections by first removing weights and then retraining the network was first shown by [Ha15]. Yu et al. [Yu17] show that by exploiting the Compressed Sparse Row format on single-instruction-multiple-data (SIMD) units of a microcontroller, pruning can be implemented by removing connections between two neurons. However, the authors learned that weight pruning on GPUs actually slows down the inference time. This deceleration is attributed to the overhead due to sparse data structures which can only be overcome by a pruning rate of 97%. However, pruning rates of more than 90% have shown to lead to a strong decrease in accuracy [Ha15; Yu17] and, therefore, weight pruning was not implemented on GPUs.

3 Weight Pruning for Deep Neural Networks on GPUs

In this work, we optimize weight pruning for the use on GPUs. GPUs have a very distinctive memory architecture, where accesses to the global memory have a high latency and the memory width is wide (e.g., 128 byte). The latency can be hidden by the massively parallel architecture of GPUs. The wide memory architecture connects the density of the information in global memory with the efficiency of memory bandwidth utilisation: In a sparse data layout where 1 out of 16 bytes is requested from memory, the bandwidth utilisation is the same as for a request of 16 bytes. Additionally, the memory architecture requires threads to access the memory in a coalesced manner.

Our approach for memory-aware weight pruning takes the distinct memory architecture of GPUs into account. Instead of pruning individual weights without considering the memory architecture, we propose to treat the weights in groups: First, the weights are organized in groups of adjacent weights with a configurable size. Then, all weights in a group are aggregated and then evaluated. All groups with an aggregated value below a defined threshold are removed from the network. The threshold is determined by the aggregated values of the groups in order to achieve a target pruning rate. Afterwards, the remaining groups are transferred to the compressed sparse row format (CSR). We use this format to be able to use sparse matrix computations in order to accelerate the computations. Sparse matrix-matrix multiplications are optimized to exploit matrices where only a small number of values is different from zero. However, the sparse formats come with some overhead.

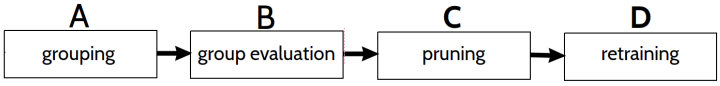

Fig. 2: The subsequent steps of our Memory-aware Weight Pruning technique.

An overview of our technique is depicted in Figure 2. First, we arrange all weights in groups according to the selected group size g in step (A). Then we evaluate the aggregated weight of each group in step (B). In step (C) we perform the actual removal of weights. First, we calculate the threshold t necessary to achieve a given pruning rate and then we remove all weights of all groups with a smaller aggregated weight. Finally, the network is retrained in step (D).

The advantage of using adjacent groups of weights is that they can be loaded at the same time in memory. This ensures that the available memory bandwidth to global memory is used efficiently. In order to be able to evaluate the significance of a group of weights we need to aggregate the weights in the group. We use the RMS (root mean square) function to calculate the aggregated weight of a group, in the same way as related work (e.g., [Yu17]). The motivation is that a high value of a weight has a strong influence on the activation of the neuron in the next layer and that high values will be further amplified by the RMS aggregation of weights.

However, we evaluated different weight aggregation functions (root mean square, arithmetic mean, median, random) and we were not able to observe significant differences in in terms of their influence on the final accuracy of the retrained network. Therefore, we assume that the selection of the groups is not as critical as it might look but the pruning rate and the retraining dictate the accuracy.

The weight matrix stored in CSR format is multiplied with a dense matrix or vector. The weight groups are selected consecutivly within a row of the matrix and with an offset that is a multiple of the group size. We show an example of the grouping step A in Equation 1. Weight matrix M is a 4×4 matrix. The round brackets indicate the weight groups with a

group size $g = 2$. The matrix contains the weights w which form the groups G. The group dimension of the matrix M is 4x2. $G_{1,1}$ is the group consisting of the of weights $w_{1,1}$ and $w_{1,2}$.

$$\mathrm{M} = \begin{bmatrix} \begin{pmatrix} w_{1,1} & w_{1,2} \end{pmatrix} \begin{pmatrix} w_{1,3} & w_{1,4} \end{pmatrix} \\ \begin{pmatrix} w_{2,1} & w_{2,2} \end{pmatrix} \begin{pmatrix} w_{2,3} & w_{2,4} \end{pmatrix} \\ \begin{pmatrix} w_{3,1} & w_{3,2} \end{pmatrix} \begin{pmatrix} w_{3,3} & w_{3,4} \end{pmatrix} \\ \begin{pmatrix} w_{4,1} & w_{4,2} \end{pmatrix} \begin{pmatrix} w_{4,3} & w_{4,4} \end{pmatrix} \end{bmatrix} = \begin{bmatrix} G_{1,1} & G_{1,2} \\ G_{2,1} & G_{2,2} \\ G_{3,1} & G_{3,2} \\ G_{4,1} & G_{4,2} \end{bmatrix} \tag{1}$$

In step B the aggregated weight of each group is calculated.

$$\mathrm{RMS} = \sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2} \qquad \text{and} \qquad M_{RMS} = \begin{bmatrix} \mathrm{RMS}(G_{1,1}) & \mathrm{RMS}(G_{1,2}) \\ \mathrm{RMS}(G_{2,1}) & \mathrm{RMS}(G_{2,2}) \\ \mathrm{RMS}(G_{3,1}) & \mathrm{RMS}(G_{3,2}) \\ \mathrm{RMS}(G_{4,1}) & \mathrm{RMS}(G_{4,2}) \end{bmatrix} \tag{2}$$

Next, a threshold is defined. The aggregated weights of the groups are sorted by value to select groups with the smallest influence (smallest absolute value). Depending on the pruning rate the threshold is selected. The groups which are below this threshold value are set to 0. In our example, after the groups have been set to 0, the matrix has the following form:

$$M_{prun} = \begin{bmatrix} \begin{pmatrix} 0 & 0 \end{pmatrix} & \begin{pmatrix} 0 & 0 \end{pmatrix} \\ \begin{pmatrix} w_{2,1} & w_{2,2} \end{pmatrix} & \begin{pmatrix} w_{2,3} & w_{2,4} \end{pmatrix} \\ \begin{pmatrix} 0 & 0 \end{pmatrix} & \begin{pmatrix} 0 & 0 \end{pmatrix} \\ \begin{pmatrix} 0 & 0 \end{pmatrix} & \begin{pmatrix} w_{4,3} & w_{4,4} \end{pmatrix} \end{bmatrix} \tag{3}$$

In step D the matrix M_{prun} is converted to the CSR format. The matrix takes the following form:

$$\mathrm{A} = \begin{bmatrix} w_{2,1} & w_{2,2} & w_{2,3} & w_{2,4} & w_{4,3} & w_{4,4} \end{bmatrix}$$

$$\mathrm{JA} = \begin{bmatrix} 0 & 1 & 2 & 3 & 2 & 3 \end{bmatrix}$$

$$\mathrm{IA} = \begin{bmatrix} 0 & 0 & 4 & 4 & 6 \end{bmatrix}$$

The new representation of the matrix consists of A, JA and IA. It is CSR format and contains only the weights of the neural network that are greater than zero. These are the weights that were selected in step B and C and will be used later.

Tab. 1: Details of our system used to conduct the results

Hardware	Model
CPU	Intel Core i5-7200U
GPU	NVIDIA Geforce GTX 950M
GPU main memory	2 GB
Software	**Version**
Ubuntu	16.04.5 LTS
CUDA	9.0.176
Python	3.5.2
Keras	2.2.4
TensorFlow	1.10.1

4 Experimental Evaluation

In this section we briefly introduce our experimental evaluation. First, we examine how to measure the execution time, then we explain how we measure the accuracy of the neural network.

4.1 Performance

As our pruning technique is optimized for fully connected layers and the inference of fully connected layers is based on matrix-matrix multiplications we conduct our performance evaluation using a matrix-matrix multiplications benchmark. All our experiments are performed using an NVIDIA Geforce GTX 950M GPU. We execute the matrix-matrix multiplication calculations in CSR format on the graphics card. The conventional calculation of matrix-matrix multiplications is called dense matrix-matrix multiplication below, and we use NVIDIA's implementation for these multiplications [Nv12].

To calculate the matrix-matrix multiplication on the GPU and to measure the execution time we use CUDA 9[Nv18]. The sparse matrix-matrix multiplications are performed with the NVIDIA's cuSPARSE library. The dense matrix-matrix multiplications are performed with the library cuBLAS.

In our benchmark two matrices of dimensions 4096×4096 and 4096×50 are multiplied with each other. The matrix sizes are chosen to achieve comparability with the work of Yu et al.[Yu17] The pruned weights are merged into the summarized format described in Section 3. For each of our performance experiments we measure the kernel execution time only, as this share of the overall execution time is the predominant part.

cuSparse offers the CSR and HYB sparse formats for calculations. For the CSR format, the library offers a matrix-matrix multiplication where the first matrix is a sparse matrix and the second matrix is a dense matrix. The HYB format is a mix of ELL format and COO format. Unfortunately, for HYB format, cuSparse offers no support for matrix-matrix multiplication but only a function for a matrix vector multiplication. For this reason, the CSR format was chosen. In the calculation of the inference of a fully connected neural network, above all, the matrix-matrix multiplication is the predominant part of work load. The addition of bias has, according to our investigations, only a minor role. We do not evaluate our pruning technique in terms of training time. The total training time of the network is increased, because the retraining time of the pruned network is added to the training time of the network.

4.2 Accuracy

In this section we describe how the accuracy of the network was determined. We use the MNIST dataset [LC19] that consists of 60,000 images of handwritten digits. Each image has a size of 28x28 pixels and can belong to 1 of 10 categories spanning the numbers between 0 and 9. The data set is randomly separated into two distinct subsets: 1) training data (54 000 images) and 2) test data for validation (6 000 images) in order to avoid over-fitting.

Tab. 2: Structure of our neural network

Layer part	Layer type	Activation function	Size
input	fully connected	ReLU	784
hidden	fully connected	ReLU	128
hidden	fully connected	ReLU	128
hidden	fully connected	ReLU	256
hidden	fully connected	ReLU	256
hidden	fully connected	ReLU	512
hidden	fully connected	ReLU	512
output	fully connected	Softmax	10

To evaluate our technique we use a neural network that consists of eight fully connected layers. This network serves the purpose to allow us to assess our technique. The structure of our network is shown in Table 2. Each of the 784 ($= 28 \times 28$) pixels represents an input of the input layer. We use the stochastic gradient descent (SGD) as optimization function.

First, we have to train the network and therefore we train the model for 3000 epochs using the training data set before we start the pruning process. In order to study the effects of the group size the weight matrices were subdivided into different group sizes during the pruning process and then the root-mean-square (RMS) is determined for each group. We implement our technique using the Keras API with the TensorFlow backend. When the aggregated weight of the groups is determined the groups are sorted by the aggregated weight. Then we set the threshold in a way such that the given pruning rate is reached. Finally, the aggregated

weights with an RMS smaller than the threshold are removed. After pruning, we retrain the neural network over 6000 epochs. The number of training epochs before and after training are chosen to match the related work [Yu17].

4.3 Performance

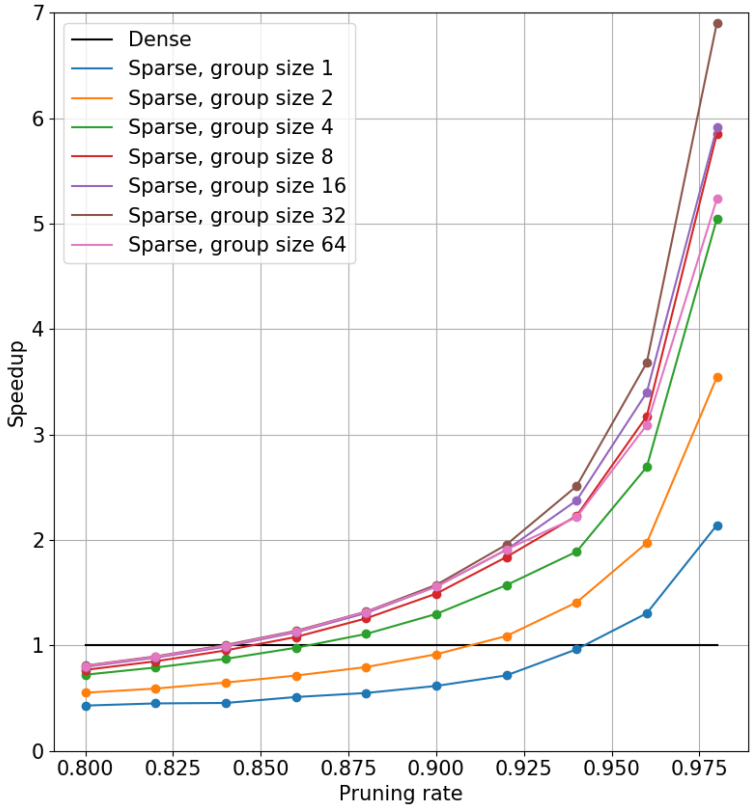

Fig. 3: Speedup of sparse matrix-matrix multiplication compared to dense matrix-matrix multiplication. The matrices have the sizes of 4096x4096 and 4096x50.

5 Results

In this section, we discuss the results of our experiments. Figure 3 shows how the performance is affected when using different pruning rates. Figure 4 shows the accuracy of different group sizes when setting the pruning rate to 90%. In Figure 5 the first two results are related to each other.

Figure 3 shows the speedup of a sparse matrix-matrix multiplication of size 4096×4096 by 4096×50 for different pruning rates between 80% and 98% and group sizes of 1, 2, 4, 8, 16, 32 and 64. The speedup is calculated by normalizing the execution time of the pruned network to the execution time of the dense network. We reproduce the work of Yu et al. and show in their results labeled *sparse, group size 1*. In our benchmark the sparse matrix-matrix multiplication at a pruning rate of 94% without grouping was as fast as the dense matrix-matrix multiplication. A similar pruning rate without grouping was reported by Yu et al. considered too large, because the accuracy would drop too much. We show that for a sparse matrix-matrix multiplication with a group size of 32, the speedup is greater than 1 at a pruning rate of 84% compared to a pruning rate of 94% for the state-of-the-art of Yu et al. In their work, the lowest reported pruning rate of 90% required more than twice the time when compared to conventional dense matrix-matrix multiplication. When we apply our grouped pruning technique, we reach a speedup of 57% achieved given the same pruning rate.

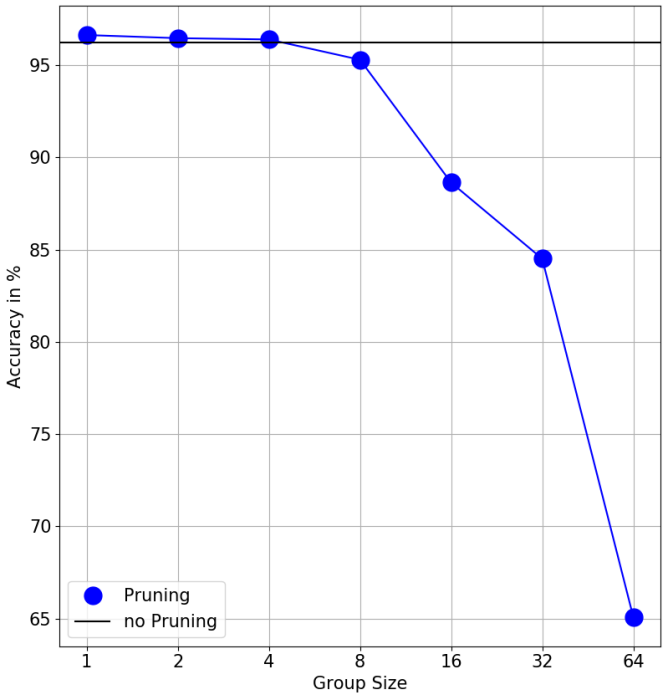

Fig. 4: Achieved accuracy per group size of the pruning technique. The pruning rate set to 90%.

In Figure 3 we show that a higher pruning rate leads to a higher speedup in the sparse matrix-matrix multiplications. The highest speedup is achieved with a group size of 32. We can attribute the speedup at least partially to the amount of coalesced memory accesses. Coalesced memory accesses are important on GPUs in order to exploit the memory bandwidth. The *Global Load Efficiency* (as reported by NVProf) increases from 67 % to

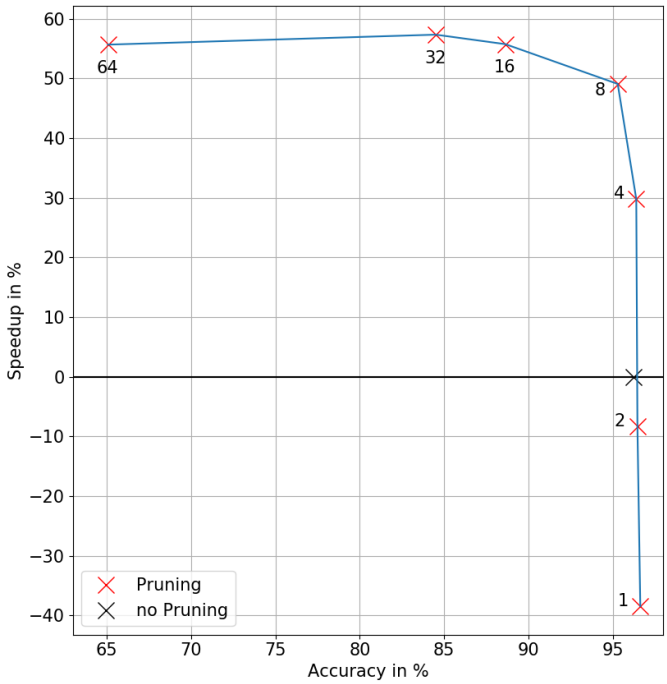

Fig. 5: Achieved accuracy in relation to the achieved speedup for different group sizes when setting the pruning rate to 90%.

82 % for a group size of 1 if the pruning rate is increased from 80 % to 98 %. From a group size of 8 the *Global Load Efficiency* rises to over 99 %.

5.1 Accuracy

In this section we discuss the accuracy achieved with different group sizes at 90% pruning rate. Figure 4 shows the accuracy of the network shown in Table 2. The horizontal line shows the accuracy of accuracy comparable to trained network that was not pruned. The blue dots show the accuracy of the networks, which were pruned with a pruning rate of 90%. The figure shows that although the pruning rate was set to 90% for all networks, the accuracy decreases the larger the group size. A sharp decline in accuracy can be observed as the group size increases. With smaller group sizes, the accuracy of the network could be maintained or even slightly improved. We assume that the improvement is the consequence of an increased training time of the network. The accuracy is higher for a small group size and many groups than for a large group size and fewer groups, as it is more likely to prune the weights that do not contribute to the activation of the neuron.

Figure 5 shows the accuracy values of the Figure 4 in relation to the GPU execution times of sparse matrix-matrix multiplications at 90% pruning rate in Figure 3. The black cross in the figure denotes the neural network without pruning. The red crosses mark speedup and accuracy of the neural network when we apply our technique and set the pruning rate to 90%. The group sizes 1, 2, 4, 8, 16, 32 and 64 of the pruning networks are written next to the respective cross of the result. It can be seen that a group size of 32, as already shown in Figure 5, offers the highest speed gain, but severely limits the accuracy of the network. Group sizes 1 and 2 even cause the neural network to perform the inference slower because the overhead generated by the CSR format is greater than the speed gain produced by exploiting Global Load Efficiency of the GPU. When aiming for an accuracy comparable to the original dense network, a group size of 8 is superior, because at this size the accuracy of the neural network is 95.30% while the speedup of 49.08% over the dense network. Sizes 1, 2 and 4 have a significantly lower speedup at a pruning rate of 90 %, as shown in the Figure 3, since the locality of the weights in the memory can not be used here.

6 Conclusion

In this paper, we propose to use memory-aware weight pruning for accelerating the inference time of deep neural networks on GPUs. Our techniques remove weights in a fully-connected layer in continuous groups of multiple weights. By aligning the weight groups to match the size of the memory architecture of current GPUs, we are able to accelerate the inference time by a factor of 1.5× for a given pruning rate of 90%. Furthermore, by using our technique we are able to lower the required pruning rate necessary to be profitable on a GPU to 84%, while state-of-the-art pruning requires a pruning rate as high as 94%. We explore how the group size affects the accuracy and what group size is optimal when given a target accuracy. In future work we will investigate different ways of determining the ranking of the weight groups, as our observation that even randomly selected weight groups result in an equal accurate network is very interesting. We will explore the design space given by different matrix sizes in the matrix-matrix multiplications. Additionally, we will be researching domain-specific sparse matrix representations in order to exploit the distinct properties of sparse neural networks. We plan to study the effects of our approach on different networks, new GPU generations and more complex applications.

References

[De13] Denil, M.; Shakibi, B.; Dinh, L.; De Freitas, N., et al.: Predicting parameters in deep learning. In: Advances in neural information processing systems. Pp. 2148–2156, 2013.

[Go18] Gong, J.; Shen, H.; Zhang, G.; Liu, X.; Li, S.; Jin, G.; Maheshwari, N.; Fomenko, E.; Segal, E.: Highly Efficient 8-bit Low Precision Inference of Convolutional Neural Networks with IntelCaffe. eprint arXiv:1805.08691v1/, 2018.

[Gu15] Gupta, S.; Agrawal, A.; Gopalakrishnan, K.; Narayanan, P.: Deep learning with limited numerical precision. In: International Conference on Machine Learning. Pp. 1737–1746, 2015.

[Ha15] Han, S.; Pool, J.; Tran, J.; Dally, W.: Learning both weights and connections for efficient neural network. In: Advances in neural information processing systems. Pp. 1135–1143, 2015.

[HMD15] Han, S.; Mao, H.; Dally, W. J.: Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. arXiv preprint arXiv:1510.00149/, 2015.

[Hu18] Huang, Q.; Zhou, K.; You, S.; Neumann, U.: Learning to Prune Filters in Convolutional Neural Networks. eprint arXiv:1801.07365v1/, 2018.

[LC19] LeCun, Y.; Cortes, C.: MNIST handwritten digit database./, 2019, URL: http://yann.lecun.com/exdb/mnist/.

[LDS90] LeCun, Y.; Denker, J. S.; Solla, S. A.: Optimal brain damage. In: Advances in neural information processing systems. Pp. 598–605, 1990.

[Li17] Li, H.; Kadav, A.; Durdanovic, I.; Samet, H.; Graf, H. P.: Pruning Filters for efficient ConvNets. ICLR/, 2017.

[Mo17] Molchanov, P.; Tyree, S.; Karras, T.; Aila, T.; Kautz, J.: Pruning Convolutional Neural Networks for resource efficient inference. ICLR/, 2017.

[Nv12] Nvidia: CUDA Toolkit 4.2 CUSPARSE Library. PG05329041v01/, 2012.

[Nv18] Nvidia: Parallele Berechnungen mit CUDA, (WWW), 2018, URL: https://www.nvidia.de/object/cuda-parallel-computing-de.html.

[Yu17] Yu, J.; Lukefahr, A.; Palframan, D.; Dasika, G.; Das, R.; Mahlke, S.: Scalpel: Customizing DNN Pruning to the Underlying Hardware Parallelism. ISCA/, 2017.