GPU Kernels for Block-Sparse Weights

Scott Gray, Alec Radford and Diederik P. Kingma
OpenAI
[scott,alec,dpkingma]@openai.com

Abstract

We're releasing highly optimized GPU kernels for an underexplored class of neural network architectures: networks with block-sparse weights. The kernels allow for efficient evaluation and differentiation of linear layers, including convolutional layers, with flexibly configurable block-sparsity patterns in the weight matrix. We find that depending on the sparsity, these kernels can run orders of magnitude faster than the best available alternatives such as cuBLAS. Using the kernels we improve upon the state-of-the-art in text sentiment analysis and generative modeling of text and images. By releasing our kernels in the open we aim to spur further advancement in model and algorithm design.

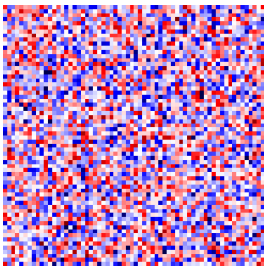
1 Introduction

Research in the field of contemporary deep learning [LeCun et al., 2015] is largely constrained by the availability of efficient GPU kernels for defining, evaluating and differentiating various model architectures. Only a small number of types of linear operations currently have efficient GPU implementations; three linear operations that currently enjoy efficient GPU implementations are dense dot products, convolutional operations and (most recently) depth-wise operations. Such operations have two variables as inputs: one input is usually a layer of network activations (corresponding to the current minibatch of datapoints), and another input that is usually the set of learned *weights* for that layer. For dense linear operations, these weights are a dense matrix or a dense higher-dimensional tensor. Two dimensions of these weights correspond to the so-called *feature* dimensions, whose lengths equal the so-called *widths* of the input and output layers of the operations. Such dense linear operations do not scale well in the feature dimensions, since the number of weights is proportional to both the number of input features, and the number of output features. Linearly scaling up the number of input and output features, results in a quadratic increase in the total number of weights, and a quadratic increase in the computational cost.

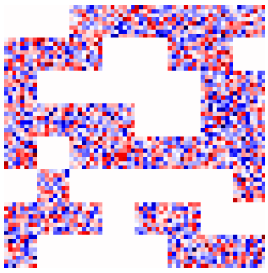
Ideally, we would have efficient operations that allow for *sparsity* in the two feature dimensions. With sparsity, we mean that the value of a subset of weights are specified to be exactly zero. If a weight is zero, then the linear operation associated with that weight can be skipped, since any value times zero equals zero. Therefore, the computational cost of sparse linear operations is only proportional to the number of non-zero weights. A problem of operations with weights with arbitrary sparsity, is that they cannot be efficiently implemented on contemporary GPUs. GPUs consist of thousands of computational cores that all operate in parallel. This restricts us to the set of operations that allow for a high degree of parallelizability, which does not include operations with arbitrary sparsity patterns.

However, we found that highly optimized *block-sparse* operations, with block sizes as small as 8×8, can still run efficiently on contemporary GPUs. See figure 1 which explains block-sparse connectivity. We introduce highly optimized GPU implementations of various block-sparse operations. The operations come in roughly two flavors: (1) block-sparse matrix multiplications, and (2) block-sparse convolutional operations. The kernels and their main documentation can be found on GitHub [1]. Please refer to this GitHub page for more information on the API.

[1] https://github.com/openai/blocksparse



Dense weights



Block-sparse weights

0	0	1	1	1	1	1	1
1	1	1	0	0	1	1	0
1	0	0	0	0	0	1	1
1	1	1	1	0	0	1	1
1	0	1	1	1	1	1	1
0	1	0	0	0	0	0	0
1	1	1	0	1	1	0	0
1	0	0	0	0	1	1	1

Corresponding sparsity pattern

Figure 1: Visualization of random dense and random block-sparse weight matrices, where white indicates a weight of zero. Our new kernels allow efficient usage of block-sparse weights in fully connected and convolutional layers, as illustrated in the middle figure. For convolutional layers, the kernels allow for sparsity in input and output feature dimensions; the connectivity is still dense in the spatial dimensions. The sparsity is defined at the level of blocks (right figure), with block size of at least 8×8. At the block level, the sparsity pattern is completely configurable. Since the kernels skip computations of blocks that are zero, the computational cost is only proportional to the number of weights, not the number of input/output features.

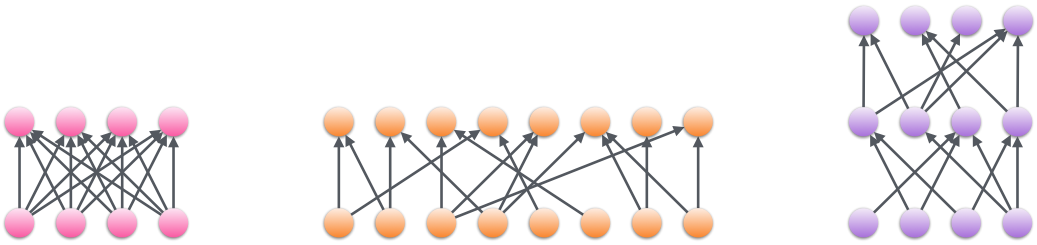


Figure 2: Dense linear layers (left) can be replaced with layers that are sparse and wider (center) or sparse and deeper (right) while approximately retaining computational cost and memory cost. Note these costs are, in principle, proportional to the number of non-zero weights (edges). The shown networks have an equal number of edges. However, the sparse and wide network has the potential advantage of a larger information bandwidth, while the deeper network has the potential benefit of fitting nonlinear functions.

Block-sparsity unlocks various research directions (see section 6). One application we explore in experiments is the widening or deepening of neural networks, while increasing sparsity, such that the computational cost remains approximately equal as explained in figure 2. In experiments we have only scratched the surface of the applications of block-sparse linear operations; by releasing our kernels in the open, we aim to spur further advancement in model and algorithm design.

2 Capabilities

The two main components of this release are a block-sparse matrix multiplication kernel and a block-sparse convolution kernel. Both are wrapped in Tensorflow [Abadi et al., 2016] ops for easy use and the kernels are straightforward to integrate into other frameworks, such as PyTorch.

Both kernels support an arbitrary block size and are optimized for 8x8, 16x16, and 32x32 block sizes. The matrix multiplication kernel supports an arbitrary block layout which is specified via a masking matrix. In addition, the feature axis is configurable. The convolution kernel supports non-contiguous input/output feature blocks of any uniform or non-uniform size specified via a configuration format (see API) though multiples of 32x32 perform best. Arbitrary dense spatial filter sizes are supported in addition to dilation, striding, padding, and edge biasing.

A variety of efficient helper ops are included for common routines such as layer and batch normalization of activations, L2 normalization of weights, dropout, activation functions, and elementwise math.

Since sparse networks allow for much larger activation tensors than dense networks, operations tend to be bandwidth bound instead of compute bound on GPU hardware. Reduced precision formats lower bandwidth significantly which helps alleviate this problem. To this end, the kernels support fp16 in addition to fp32 with additional compact formats such as bfloat16 in active development.

3 Benchmarks

3.1 Performance (GFLOPS) compared to cuBLAS and cuSPARSE kernels

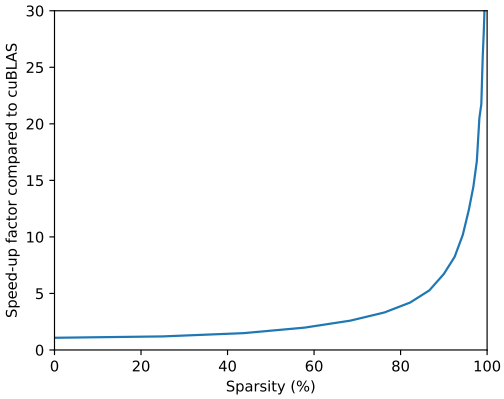


Figure 3: Empirical speed-ups, in terms of relative GFLOPS, of block-sparse matrix multiplication with a 12288×12288 weight matrix, a minibatch size of 32, and a block size of 32. We compare against cuBLAS (CUDA 8) matrix multiplication. Other baselines typically faired worse than cuBLAS.

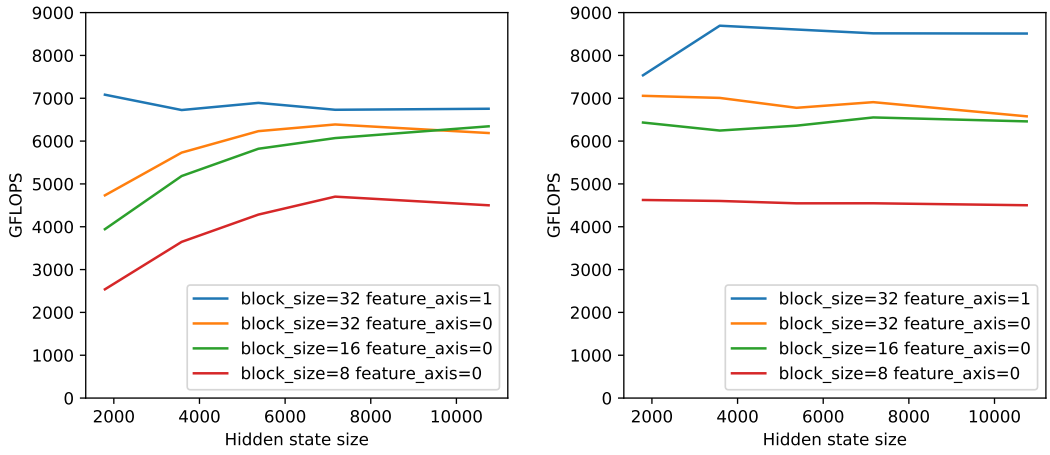
In order to verify the efficiency of our proposed kernels, we compare against three baseline techniques for linear layers with block-sparse weights. For all cases, we tested on a NVIDIA Pascal Titan X GPU, with minibatch size 32 and block size 32×32.

The first baseline technique is the naïve use of cuBLAS kernels with sparse weight matrices. Since this technique does not 'skip' blocks of weights whose values are 0, the computational complexity is proportional to the total number of entries in the matrix, not the number of non-zero blocks. Therefore, this technique performs a lot of unnecessary operations. See figure 3 for the relative speed of our kernel, compared to this technique. For higher degrees of sparsity, we see as expected a speedup factor close to $\frac{1}{1-s/100}$, where s is the sparsity percentage.

We also compared against baselines of (1) block-sparse matrix multiplication through performing a sequence of small per-block matrix multiplications with cuBLAS, and (2) block-sparse matrix multiplication using the cuSPARSE library. Unlike the previous baseline, the computational complexities of these methods are only proportional to the number of non-zero blocks. Still, in our experiments, these baselines faired worse than the previously baseline of naïve usage of cuBLAS; the number of GFLOPS did not not exceed about 50, regardless of the degree of sparsity. Our kernels typically performed one or two orders of magnitude faster in terms of GFLOPS.

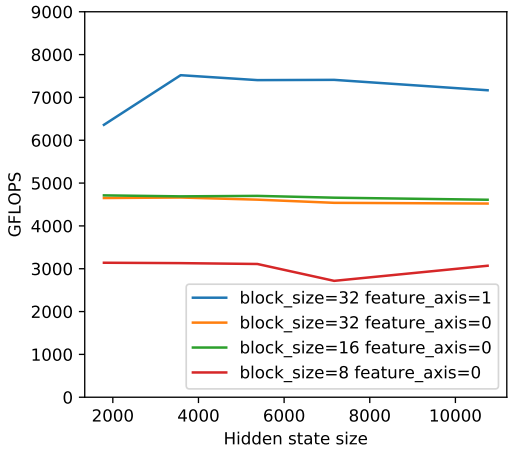
3.2 Effect of block size, features axis and hidden state size

We benchmarked the performance of our kernels, in terms of GFLOPS, as a function block size, features axis and hidden state size; see figure 4. In each experiment, we kept the total number of parameters fixed. This experiment was performed on a NVIDIA P100 GPU, with a small-world LSTM with about 3 million parameters, with a hidden state size ranging from 1792 to 10752, corresponding to 0% to 97% sparsity. The evaluation was done with a minibatch size of 64; this



(a) *fprop/bprop* operation

(b) *update* operation with group size 8



(c) *update* operation with group size 1

Figure 4: (a-c): Performance of elementary operations in our proposed kernels in terms of GFLOPS as a function of block size, feature axis and the hidden state size. See section 3.

size often performs best due to reduced cache dilution compared to larger minibatch sizes. The connectivity pattern is generated with the Watts-Strogatz algorithm, with 20% random long range connections, but performance with Barabási-Albert connectivity is close.

The operation with feature_axis=1 corresponds to an assembly-optimized kernel, and is essentially the same kernel as the *openai-gemm* kernel, now also used in cuBLAS for tile size 32x32. This kernel clearly outperforms the kernel for feature axis 0, but does not work for Kepler and Volta GPUs.

Figure 4a shows performance of the *fprop/bprop* operation, which compute forward activations, and compute gradients w.r.t. the forward activations respectively. Since *bprop* is the transpose of the *fprop* operation, and transposes can be done in-place, the two operations have identical performance. Note that, perhaps somewhat surprisingly, in this experiment a higher degree of sparsity generally leads to better performance in GFLOPS. This is due to the higher parallelizability of the accumulation operation in case of larger hidden state sizes.

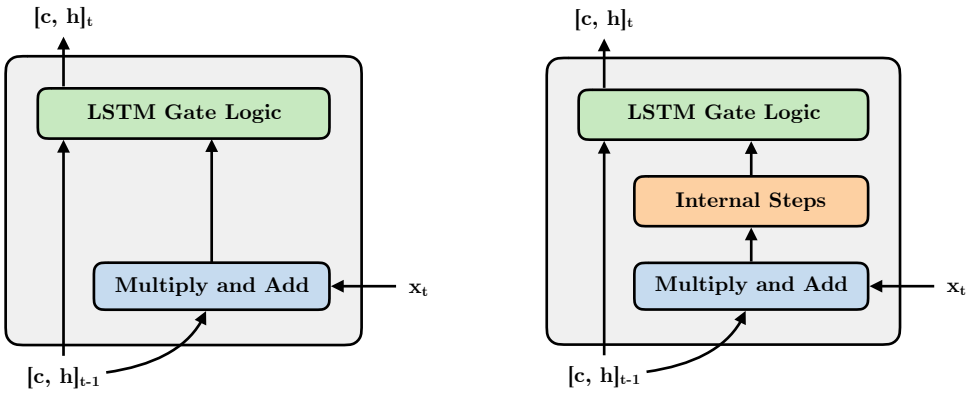
In figures 4b and 4c we benchmarked the *update* operation, which computes derivatives w.r.t. the block-sparse weights. Note that if a weight matrix is re-used multiple times in a computational graph, such as in an LSTM, this operation can be grouped, i.e. performed in parallel across multiple timesteps. Comparing figure 4b with 4c, it is clear that the grouping leads to substantial improvement

in GFLOPS. Grouped update operations are easy to perform, since the kernels take lists of (activation, gradients) as input, avoiding the requirement of pre-concatenation.

4 Experiments

Our new kernels open up a large space of new possibilities for learning algorithms and neural network architectures. In this section ,we experiment with some relatively simple block-sparse patterns with fixed sparse topology. Note that the kernels in principle allow for much more exotic methods, such as a learned sparsity structure, and more exotic models; we leave this for future work.

4.1 LSTMs with Deep Updates



(a) Multiplicative LSTM architecture

(b) Our LSTM architecture with deep updates

Figure 5: In experiments, we use an LSTM architecture with deep updates (right), where all linear operations have block-sparse connectivity with small-world topology. The architecture can be seen as an evolution of the multiplicative LSTM (left) [Krause et al., 2016].

The block-sparse kernels allow us to efficiently implement LSTMs with block-sparsity connectivity. With sparse connectivity, we mean that the linear operations in a LSTM are replaced by block-sparse linear operations. Sparse connectivity allows us to, for example, widen the network without increasing the number of parameters.

One appealing property of LSTMs with densely connected recurrent connections is that the value of each activation (neuron) at timestep t is a function of all hidden activations, and all inputs, at timestep $(t-1)$. In other words, information fully *mixes* between timesteps. This is not true for a LSTM with naïve block-sparse connectivity: since not all neurons are directly connected, information would not fully mix between timesteps. A similar observation was made in [Wen et al., 2017] and motivated their structured sparsity approach.

We choose to tackle this problem with the introduction of a sparse multi-layer network within each LSTM cell; see figure 5 for an illustration, and algorithm 1 in the appendix for pseudo-code of this LSTM with deep updates. Given the right block-sparse connectivity, such networks can fully mix within a relatively small number of layers when combined with a small-world connectivity, as we will explain in section4.2. The LSTM architecture builds on the multiplicative LSTM (mSLTM) architecture proposed in [Krause et al., 2016], and utilizes layer normalization [Ba et al., 2016] for improved efficiency.

Adding internal depth is a good way to increase parameter efficiency, even in the dense case; see figure 8 in the appendix. Performance seems to saturate after about 4 internal steps.

4.2 Small-World Networks

In the choice of sparse connectivity, we take inspiration from the field of *small-world networks*, as we will now explain. A network is defined to be a small-world network [Watts, 1999] if the following two properties hold.

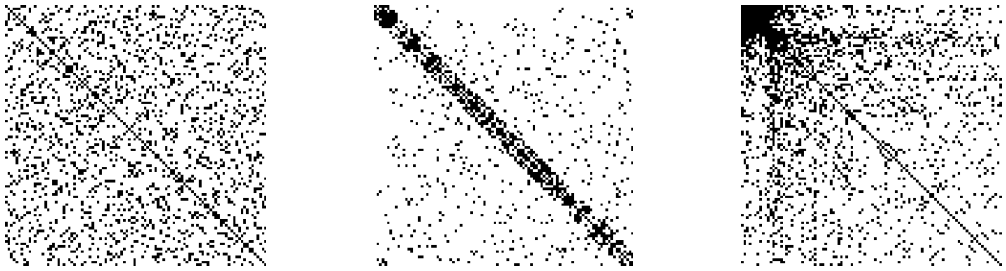


Figure 6: Visualization of adjacency matrices of random, Watts-Strogatz (WS) with a 1-Dimensional ring lattice, and Barabási-Albert (BA) networks. The latter two correspond to small-world networks, defined as having both a large clustering coefficient, and short average path length (see section 4.2). The purely random network, shown left, is not a small-world network, as it lacks clustering.

1. The *clustering coefficient* is not small. The clustering coefficient is a measure of locality of connectivity, and is high when the graph contains many cliques or near-cliques: subnetworks that are almost fully interconnected. Note that RNNs constructed with block-sparse connectivity automatically have this property.
2. The *average path length* L between nodes (neurons) scales only logarithmically with the total number of nodes/neurons N, i.e. $L \propto \log N$. The path length between nodes equals the length of the shortest path between these nodes. A short average path length leads to rapid mixing of information.

A well-known example of small-world networks is the human social network [Watts, 2004]: our friends, family and acquaintances often also know eachother (high clustering), but we are also on average only about 'six handshakes away' from a random other person on earth. Another example is the human brain, whose anatomical and functional networks often also exhibit small-worldness [Bassett and Bullmore, 2006].

Some algorithms for generating graphs with small-world properties are:

1. Random block-sparse connectivity. Note that a plain block-sparse structure with non-trivially sized blocks, automatically has some degree of clustering. This results in the simplest type of small-world RNN.
2. The Watts-Strogatz (WS) model [Watts and Strogatz, 1998]. The algorithm that construct WS graphs starts out with a K-dimensional ring lattice with a dense purely local connectivity; every node is connected to every other node within a certain distance within the lattice. Then a random subset ($k\%$) of all connections is replaced with a random connection. The other $(100 - k)\%$ local connections are retained.
3. The Barabási-Albert (BA) model [Barabási and Albert, 1999]. The algorithm that constructs such graphs begins with an initial densely connected network with m_0 nodes. Then, new nodes are added on at a time, each new node connected to $m \leq m_0$ existing nodes, where the probability that a new node connects to a particular existing node i is proportional to the degree (the number of connections) of that existing node. This leads to a power law degree distribution, with a very small number of nodes with a very large degree of connectivity ('the rich get richer').

Our block-sparse GPU kernels allow us to implement models with WS and BA connectivity on the block level. See figure 6 for an illustration of the types of connectivity.

Neural networks with small-world connectivity allow us to train wider networks without incurring a quadratic increase in the number of parameters. This let us scale RNNs to very large states. Since the average path length between neurons scales only as the logarithm of the total number of neurons, even very large networks are fully connected within a short number of timesteps. As a result, information has the ability to mix rapidly within the network.

4.3 Small-World LSTMs

We trained LSTMs with deep updates and small-world block-sparse connectivity, which we refer to as *Small-World LSTMs*. For a large scale experiment we follow the setup of [Radford et al., 2017] and train byte-level generative models on the Amazon Reviews corpus [McAuley et al., 2015]. Due to more efficient implementations, models are trained for four epochs instead of only one. Batch size is also increased by 4x resulting in an equivalent amount of total updates. As in other experiments, we train models with nearly equivalent parameter counts and the same hyper-parameters, comparing dense weight matrices with a block-sparse variant. A dense model with a state size of 3168 is trained. The sparse model uses a Barabási-Albert connectivity pattern with an effective sparsity of ~97% and a state size of 18432. The dense model reaches 1.058 bits per byte - already a significant improvement over the 1.12 bits per byte previously reported on this dataset. The sparse model improves this further, reaching **1.048** bits per byte.

4.4 Binary Sentiment Representation Learning

In addition to the generative results above, we also compare the usefulness of these models for the task of binary sentiment classification in figure 7. We follow the semi-supervised methodology established by [Kiros et al., 2015] which trains a task-specific supervised linear model on top of a more general purpose unsupervised representation. Due in part to the much higher feature dimensionality of the sparse model increasing the effective capacity of the linear model, it outperforms the dense model on all sentiment datasets. Of particular note, our sparse model improves the state of the art on the document level IMDB dataset from 5.91% error [Miyato et al., 2016] to 5.01%. This is a promising improvement compared to [Radford et al., 2017] which performed best only on shorter sentence level datasets.

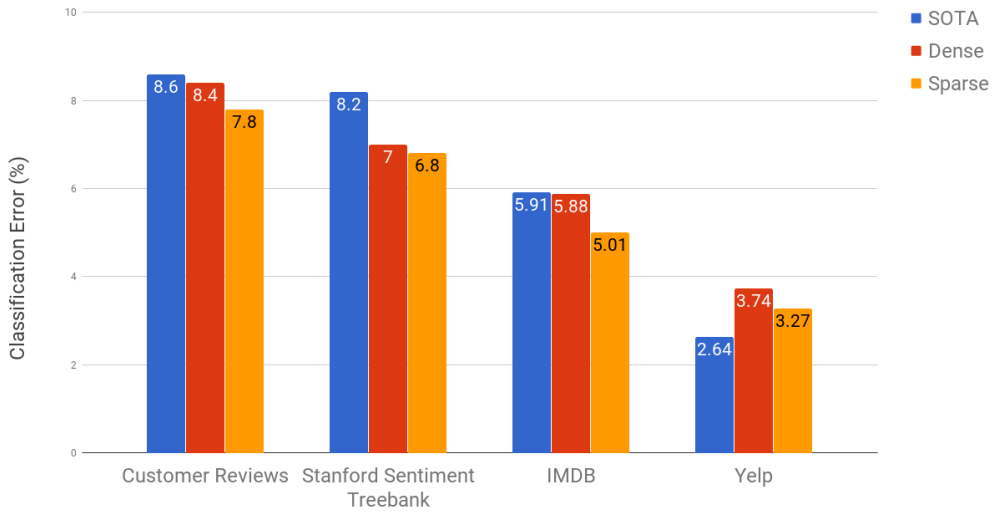


Figure 7: Binary sentiment classification error (%) of linear models trained on the features of dense and sparse generative models with approximately equivalent total parameter counts. SOTA for the Customer Reviews and Stanford Sentiment Treebank datasets from [Radford et al., 2017], for IMDB [Miyato et al., 2016], for Yelp [Johnson and Zhang, 2017].

4.5 Block-Sparse Convolutional Networks

Finally, we tested whether replacement of dense (regular) convolutional kernels with block-sparse kernels improves results in a generative modeling benchmark. In order to maximize fairness of comparison, we took a pre-existing implementation of a SOTA model, and kept all hyper-parameters (including those for optimization) unchanged, while sparsifying and deepening the model such that the total number of parameters is approximately unchanged. Specifically, we took the openly available implementation of the PixelCNN++ [Salimans et al., 2017] generative model, and replaced the regular convolutions with block-sparse convolution with a block-diagonal structure. This is also known

as *grouped convolution* [Zhang et al., 2017, Xie et al., 2016]. Similar to [Zhang et al., 2017], we added a shuffle operator after every block-sparse convolution. In our case, shuffling has no additional computational or memory cost, since it can be merged with the convolution operation, simply by doing shuffling of the sparsity structure.

We found that increasing the depth of each stage of the model by a factor 2 or 4, while increasing the sparsity so as to keep the total number of parameters approximately constant, leads to increasingly better performance in terms of the bits per dimension (bpd). With an increase of the depth by a factor 4, this resulted in **2.90** bpd, which is (to the best of our knowledge) the best reported number in the literature so far.

These results are consistent with findings by [Zhang et al., 2017] and [Xie et al., 2016], who found that similarly grouped convolution led to improvements in classification error in supervised tasks. For extensive experimental evidence that block-sparse convolutions can significantly improve results, we refer to these previous publications.

5 Related Work

There is extensive evidence in the literature that architectures with block-sparse linear operations can substantially improve results.

In [Xie et al., 2016], for example, a novel convolutional network architecture called *ResNeXt* was proposed using block-sparse convolutions, improving upon the state-of-the-art. Later, the ShuffleNet [Zhang et al., 2017] used block-sparse convolutions as well, this time in combination with a shuffle operation, like we did in our convolutional experiments. These architecture can potentially greatly benefit from efficient block-sparse GPU kernels.

Depthwise separable convolutions [Simonyan and Zisserman, 2014] can be viewed as block-sparse convolutions with a block-diagonal connectivity matrix, and block size 1. Depthwise separable convolutions have been used to advance the state of the art in image classification in various publications, starting in [Simonyan and Zisserman, 2014], later in the Xception [Chollet, 2016] and MobileNet [Howard et al., 2017] architectures.

One of our motivations for block-sparsity in RNNs; larger state sizes without significantly increased computational or parameter costs, has been successfully addressed in numerous ways by other approaches. The projection LSTM of [Sak et al., 2014] allows for more efficient usage of high dimensional states by reducing the number of parameters and computation in the hidden to hidden transition of an LSTM and has been scaled up to achieve state of the art results in large scale language modeling [Jozefowicz et al., 2016]. This work has been further improved by [Kuchaiev and Ginsburg, 2017] who explore additional factorization methods for LSTMs.

In contrast to our small-world LSTM approach which trains with a fixed block-level sparsity pattern, more flexible methods which learn structured sparsity in RNNs have been explored. Group lasso regularization [Yuan and Lin, 2006] is a popular technique which by appropriate definition of a group encourages structured sparsity. Closely related to our work, [Narang et al., 2017] used carefully scheduled thresholding and group lasso on blocks of weight matrices to learn block-sparse RNNs. [Wen et al., 2017] achieved unit level pruning via group lasso regularization of "Intrinsic Sparse Structure weight groups" - which correspond to the rows and columns of LSTM weight matrices for a specific unit.

We are far from the first to apply internal steps in a recurrent neural network. In [Zilly et al., 2016] and [Graves, 2016], for example, RNNs were trained with multiple internal steps per external timestep.

6 Research Directions

There remain a large number of unexplored research directions and potential applications of the block-sparse kernels. Here we list some open questions and suggestions for future research.

- Often, a large percentage of the weights in neural networks can be pruned after training has finished, as shown by various recent work summarized in [Cheng et al., 2017]. Typically these results could not be translated into wall-clock speedups, since there was an absence of

GPU kernels that could leverage sparsity. How much wall-clock time speed-up is possible at inference time, when using block-wise pruning of weights, together with block-sparse kernels?

- In biological brains, the sparse structure of the network is partially determined during development, in addition to connection strengths. Can we do something similar in artificial neural networks, where we use gradients to not only learn the connection weights, but also the optimal sparsity structure? A recent paper proposed a method for learning block-sparse recurrent neural networks [Narang et al., 2017], and we recently proposed an algorithm for L0 regularization in neural networks [Louizos et al., 2017], which can be used towards this end.
- We trained LSTMs with tens of thousands of hidden units, leading to better models of text. More generally, sparse layers make it possible to train models with huge weight matrices but the same number of parameters and the same computational cost as their smaller dense counterparts. What are application domains where this will make the most difference to performance?

7 Conclusion

We released highly optimized GPU kernels for gradient-based learning and inference in neural networks with block-sparse weights. In benchmarking experiments, we found that our GPU kernels indeed work much more efficiently than alternative kernels that are not optimized for block-sparse weights. We use the kernels to implement small-world LSTMs, which allow us to scale up to much wider states than typically used in LSTMs. We compared the representations (learned generatively on Amazon reviews data) of a dense network [Radford et al., 2017] with the wider and sparse variant, in terms of their usefulness for classifying sentiment. We found that the wider state indeed helped identify sentiment, leading to state-of-the-art results on various sentiment classification benchmarks. The bits-per-character results on the Amazon reviews dataset are also the best reported in the literature so far. We also saw improvements in the bits-per-dimension performance in generative modeling of CIFAR-10, when using sparse layers. Much is left to be explored in the space of block-sparse neural networks, and we have listed some potentially fruitful directions for future research.

Acknowledgments. We would like to thank Nvidia Corporation for their generous gift of a DGX-1 GPU machine, which was crucial for training our large scale block-sparse LSTMs. We would also like to thank Jack Clark, Jonas Schneider, Greg Brockman, Ilya Sutskever and Erich Elsen for their help leading up to this release.

References

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *science*, 286 (5439):509–512, 1999.

Danielle Smith Bassett and ED Bullmore. Small-world brain networks. *The neuroscientist*, 12(6): 512–523, 2006.

Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. A survey of model compression and acceleration for deep neural networks. *arXiv preprint arXiv:1710.09282*, 2017.

François Chollet. Xception: Deep learning with depthwise separable convolutions. *arXiv preprint arXiv:1610.02357*, 2016.

Alex Graves. Adaptive computation time for recurrent neural networks. *arXiv preprint arXiv:1603.08983*, 2016.

Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

Rie Johnson and Tong Zhang. Deep pyramid convolutional neural networks for text categorization. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, volume 1, pages 562–570, 2017.

Rafal Jozefowicz, Oriol Vinyals, Mike Schuster, Noam Shazeer, and Yonghui Wu. Exploring the limits of language modeling. *arXiv preprint arXiv:1602.02410*, 2016.

Ryan Kiros, Yukun Zhu, Ruslan R Salakhutdinov, Richard Zemel, Raquel Urtasun, Antonio Torralba, and Sanja Fidler. Skip-thought vectors. In *Advances in neural information processing systems*, pages 3294–3302, 2015.

Ben Krause, Liang Lu, Iain Murray, and Steve Renals. Multiplicative lstm for sequence modelling. *arXiv preprint arXiv:1609.07959*, 2016.

Oleksii Kuchaiev and Boris Ginsburg. Factorization tricks for lstm networks. *arXiv preprint arXiv:1703.10722*, 2017.

Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

Christos Louizos, Max Welling, and Diederik P. Kingma. Learning sparse neural networks through l0 regularization. *arXiv preprint arXiv:1712.01312*, 2017.

Julian McAuley, Rahul Pandey, and Jure Leskovec. Inferring networks of substitutable and complementary products. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 785–794. ACM, 2015.

Takeru Miyato, Andrew M Dai, and Ian Goodfellow. Adversarial training methods for semi-supervised text classification. *arXiv preprint arXiv:1605.07725*, 2016.

Sharan Narang, Eric Undersander, and Gregory Diamos. Block-sparse recurrent neural networks. *arXiv preprint arXiv:1711.02782*, 2017.

Alec Radford, Rafal Jozefowicz, and Ilya Sutskever. Learning to generate reviews and discovering sentiment. *arXiv preprint arXiv:1704.01444*, 2017.

Haşim Sak, Andrew Senior, and Françoise Beaufays. Long short-term memory recurrent neural network architectures for large scale acoustic modeling. In *Fifteenth Annual Conference of the International Speech Communication Association*, 2014.

Tim Salimans, Andrej Karpathy, Xi Chen, Diederik P Knigma, and Yaroslav Bulatov. Pixelcnn++: A pixelcnn implementation with discretized logistic mixture likelihood and other modifications. In *International Conference on Learning Representations (ICLR)*, 2017.

Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

Duncan J Watts. *Small worlds: the dynamics of networks between order and randomness*. Princeton university press, 1999.

Duncan J Watts. *Six degrees: The science of a connected age*. WW Norton & Company, 2004.

Duncan J Watts and Steven H Strogatz. Collective dynamics of'small-world'networks. *nature*, 393 (6684):440, 1998.

Wei Wen, Yuxiong He, Samyam Rajbhandari, Wenhan Wang, Fang Liu, Bin Hu, Yiran Chen, and Hai Li. Learning intrinsic sparse structures within long short-term memory. *arXiv preprint arXiv:1709.05027*, 2017.

Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. Aggregated residual transformations for deep neural networks. *arXiv preprint arXiv:1611.05431*, 2016.

Ming Yuan and Yi Lin. Model selection and estimation in regression with grouped variables. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 68(1):49–67, 2006.

Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices. *arXiv preprint arXiv:1707.01083*, 2017.

Julian Georg Zilly, Rupesh Kumar Srivastava, Jan Koutník, and Jürgen Schmidhuber. Recurrent highway networks. *arXiv preprint arXiv:1607.03474*, 2016.

A Appendix

Algorithm 1: Pseudo-code of one step in our LSTM architecture with deep updates. Every `linear()` operation consists of a dense or block-sparse linear operation followed by *layer normalization* Ba et al. [2016], which includes (as usual) a bias and gain operation. The parameters $\boldsymbol{\theta}$ stand for the weights, biases and gains of each linear layer. We found that layer normalization resulted in superior stability and performance. Note that due to the nonlinear layer normalization operation inside `linear()`, it is strictly speaking not a linear operation.

Hyper-parameter: $D \geq 2$ ▷ Depth of internal network
Input: $\mathbf{c}, \mathbf{h}$ ▷ Current hidden state (returned from previous timtestep)
Input: $\mathbf{x}$ ▷ Current observation
Parameters: $\{\boldsymbol{\theta}_{\mathbf{mx}}, \boldsymbol{\theta}_{\mathbf{mh}}, \boldsymbol{\theta}_{\mathbf{ax}}, \boldsymbol{\theta}_{\mathbf{ah}}, \boldsymbol{\theta}_3, ..., \boldsymbol{\theta}_N, \boldsymbol{\theta}_{\mathbf{i}}, \boldsymbol{\theta}_{\mathbf{f}}, \boldsymbol{\theta}_{\mathbf{o}}, \boldsymbol{\theta}_{\mathbf{u}}\}$ ▷ (Explanation above)

`// Multiplicative and additive layers:`
$\mathbf{h} \leftarrow \texttt{linear}(\mathbf{x}, \boldsymbol{\theta}_{\mathbf{mx}}) \odot \texttt{linear}(\mathbf{h}, \boldsymbol{\theta}_{\mathbf{mh}})$ ▷ First internal layer (multiplicative step)
$\mathbf{h} \leftarrow \texttt{ReLU}(\texttt{linear}(\mathbf{x}, \boldsymbol{\theta}_{\mathbf{ax}}) + \texttt{linear}(\mathbf{h}, \boldsymbol{\theta}_{\mathbf{ah}}))$ ▷ Second internal layer (additive step)

`// Additional internal depth:`
for $j \leftarrow 3$ **to** D **do**
  $\mathbf{h} \leftarrow \texttt{ReLU}(\texttt{linear}(\mathbf{h}, \boldsymbol{\theta}_j))$ ▷ Additional internal layers
end

`// Compute LSTM gates:`
$\mathbf{i} \leftarrow \texttt{sigmoid}(\texttt{linear}(\mathbf{h}, \boldsymbol{\theta}_{\mathbf{i}}))$ ▷ Input gate
$\mathbf{f} \leftarrow \texttt{sigmoid}(\texttt{linear}(\mathbf{h}, \boldsymbol{\theta}_{\mathbf{f}}))$ ▷ Forget gate
$\mathbf{o} \leftarrow \texttt{sigmoid}(\texttt{linear}(\mathbf{h}, \boldsymbol{\theta}_{\mathbf{o}}))$ ▷ Output gate

`// Apply LSTM update:`
$\mathbf{c} \leftarrow \mathbf{f} \odot \mathbf{c} + \mathbf{i} \odot \texttt{tanh}(\texttt{linear}(\mathbf{h}, \boldsymbol{\theta}_{\mathbf{u}}))$ ▷ Update of hidden state $\mathbf{c}$
$\mathbf{h} \leftarrow \mathbf{o} \odot \texttt{tanh}(\mathbf{c})$ ▷ Update of hidden state $\mathbf{h}$

Return $\mathbf{c}, \mathbf{h}$ ▷ Return new hidden state

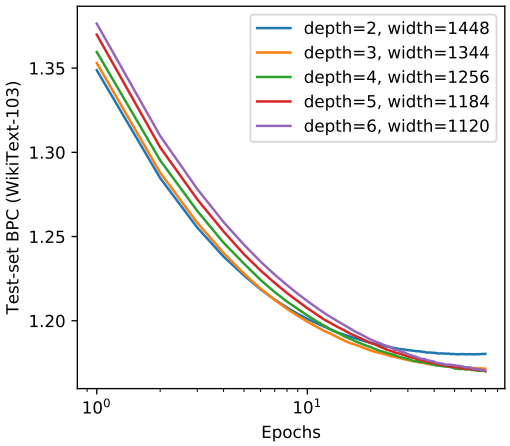


Figure 8: Training curves of our LSTM with deep updates and *dense* connectivity, as a function of the internal depth and the network width. The network width was chosen in order to closely match the number of parameters of the network with an internal depth of 2.