TIRAMISU: A Polyhedral Compiler for Dense and Sparse Deep Learning

Riyadh Baghdadi
MIT
baghdadi@mit.edu

Abdelkader Nadir Debbagh
ESI
fa_debbagh@esi.dz

Kamel Abdous
ESI
fk_abdous@esi.dz

Benhamida Fatima Zohra
ESI
f_benhamida@esi.dz

Alex Renda
MIT
renda@csail.mit.edu

Jonathan Elliott Frankle
MIT
jfrankle@mit.edu

Michael Carbin
MIT
mcarbin@csail.mit.edu

Saman Amarasinghe
MIT
saman@csail.mit.edu

Abstract

In this paper, we demonstrate a compiler that can optimize *sparse and recurrent neural networks*, both of which are currently outside of the scope of existing neural network compilers (sparse neural networks here stand for networks that can be accelerated with sparse tensor algebra techniques). Our demonstration includes a mapping of sparse and recurrent neural networks to the polyhedral model along with an implementation of our approach in TIRAMISU, our state-of-the-art polyhedral compiler. We evaluate our approach on a set of deep learning benchmarks and compare our results with hand-optimized industrial libraries. Our results show that our approach at least matches Intel MKL-DNN and in some cases outperforms it by $5\times$ (on multicore-CPUs).

1 Introduction

With the increasing need for efficient deep learning, there is a surge in hardware and compiler research, not only because compilers improve developer productivity by generating code for the new deep learning hardware accelerators, but also because compilers can significantly optimize deep learning computations (e.g., through operator fusion [33]).

Generating high performance code for deep learning requires complex code and data layout transformations, management of complex memory hierarchies, and the ability to take advantage of complex low level hardware features. While state-of-the-art deep learning compilers can optimize efficiently neural networks with acyclic data-flow graphs (feed-forward neural networks), they still have limitations in optimizing recurrent and sparse neural networks.

In this paper, we demonstrate a compiler that can optimize *sparse and recurrent neural networks* [1]. We implement our approach in TIRAMISU [6], our state-of-the-art polyhedral compiler. TIRAMISU takes a high level representation of the program (pure algorithm and a set of scheduling commands), applies the necessary code transformations, and generates highly-optimized code for the target architecture. It uses the polyhedral representation internally, which provides many advantages such as the ability to apply complex loop and data layout transformations and the ability to express programs

[1]Sparse neural networks in this context mean neural networks that can be accelerated with sparse tensor algebra techniques

that have non-rectangular iteration spaces or that have cycles in their data flow graphs. TIRAMISU relies on the use of scheduling commands, therefore it avoids many limitations that fully automatic compilers have. TIRAMISU has two unique features in the area of deep learning: (1) it introduces the first DNN (Deep Neural Network) compiler that exploits weight sparsity; and (2) it can express and optimize general RNNs (Recurrent Neural Networks). In this paper, we will demonstrate TIRAMISU by generating code for multicore CPUs.

Exploiting weight sparsity in deep neural networks (DNNs) is a promising direction for accelerating deep learning. The weights of a neural network can be made sparse using network pruning [22, 18], a technique to sparsify neural networks by removing unnecessary structure from the neural network while minimizing the loss in accuracy. Two families of network pruning techniques exist: pruning to obtain structured sparsity (e.g., by dropping convolutional filters [23]) and pruning to obtain unstructured sparsity (e.g., by dropping individual weights or connections in the neural network [18]). While structured sparsity is easy to accelerate, unstructured sparsity techniques can find much sparser networks with equivalent accuracy. State-of-the-art unstructured network pruning techniques [20, 12] can prune a ResNet-50 trained on ImageNet by 80% without any loss in accuracy and a VGG-19 trained on CIFAR-10 by 99% [11]. State-of-the-art DNN compilers however do not exploit such unstructured sparsity, due to fine-grained sparsity patterns being more difficult to accelerate, and therefore do not realize the performance gains from reduced computation and memory accesses.

In this paper, we make the following *contributions*:

- We introduce the first DNN compiler that generates efficient code for neural networks with sparse weights; In particular, TIRAMISU is the first to show that deep neural networks with unstructured weight sparsity can be accelerated by compilers;
- We introduce a DNN compiler that can express and optimize the general form of RNNs (where the number of RNN unrolling factor is unknown at compile time);
- We evaluate our compiler on a set of deep learning benchmarks and compare it with the Intel MKL-DNN library (on multicore-CPU). We show that TIRAMISU can generate efficient code that matches or outperforms Intel MKL-DNN by up to $5\times$.

2 The TIRAMISU Embedded DSL

TIRAMISU is a domain-specific language (DSL) embedded in C++. It provides a pure C++ API that allows users to write a high level, architecture-independent algorithm and a set of scheduling commands that guide code generation. TIRAMISU is integrated in high level deep learning frameworks such as Pytorch and therefore can be used transparently by end-users. It can also be generated by any other similar high level framework or DSL.

The first part of a TIRAMISU program specifies the algorithm without specifying loop optimizations (when and where the computations occur) or data-layout (how data should be stored in memory). The second part of the program provides the schedule, which specifies how the program should be optimized (vectorization, tiling, fusion, ...) and how the results of computations should be stored. The following code shows an example of a convolution algorithm written in TIRAMISU.

```
1 // Declare the iterators.
2 var n(0, batch), fout(0, out_features), fin(0, in_features);
3 var y(1, H-1), x(1, W-1), k0(0, 3), k1(0, 3);
4
5 // Algorithm.
6 conv(n, fout, y, x) +=
7     weights(fout, fin, y, x) * input(n, fin, y+k0, x+k1);
```

The iterators in line 2 define the iteration domain of conv (i.e., loop bounds). The algorithm is semantically equivalent to the following code.

```
for (n in 0..batch)
 for (fout in 0..out_features)
  for (y in 1..H-1)
   for (x in 1..W-1)
    for (fin in 0..in_features)
     for (k0 in 0..3)
      for (k1 in 0..3)
       conv[n, fout, y, x] += weigths[fout, fin, y, x] * input[n, fin, y+k0, x+k1];
```

The following code shows an example of scheduling commands (optimization commands) that can be applied on the previous convolution kernel. These commands parallelize the loop `n`, interchange the loops `fin` and `fout` and vectorize the loop `fout` by a vector length of 8.

```
conv.parallelize(n);
conv.interchange(fin, fout);
conv.vectorize(fout, 8);
```

Neural Network Optimizations Neural network optimizations applied by TIRAMISU include operator fusion, loop skewing, parallelization, multi-level tiling, loop reordering, loop unrolling, vectorization, array packing [14], register blocking, data prefetching, full/partial tile separation and tuning optimization parameters to the target architecture (e.g., choosing tile sizes or loop unrolling factors that are optimal for the target machine using auto-tuning [3]).

TIRAMISU has two unique neural network optimizations: (1) optimizing sparse convolutions (weight sparsity); and (2) optimizing RNNs (Recurrent Neural Networks). In the next section we will provide more details about how does TIRAMISU support these two optimizations.

3 Optimizing Sparse Neural Networks

Network	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
VGG-16	49.5%	34.6%	77.7%	79.5%	77.1%	65.9%	45.7%	24.2%	5.8%	1.0%
ResNet-20	61.3%	22.2%	24.0%	23.8%	21.3%	27.6%	19.4%	26.8%	20.3%	16.1%
Network	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	
VGG-16	0.2%	0.2%	0.3%	0.4%	0.7%	1.0%	N/A	N/A	N/A	
ResNet-20	12.4%	16.3%	11.0%	15.7%	13.0%	11.3%	9.2%	10.0%	2.1%	

Table 1: Density across conv layers in a pruned ResNet-20 and VGG-16

Modern CNNs for vision tend to be significantly overparameterized, imposing much higher memory and computational requirements than necessary for the task [18]. However, it is typically not possible to simply reduce the model size by using smaller models to begin with: small models trained from scratch do not reach the same accuracy as large models which are trained then sparsified [24]. Instead, the smallest models are obtained through *unstructured* pruning techniques: training a full model, then pruning individual weights from that model using some heuristic in order to create the most accurate model at a given sparsity level [18].

In this paper, we evaluate on networks obtained through a technique based on the Lottery Ticket Hypothesis [11] (although support for sparse weights in TIRAMISU is general and does not depend on the patterns of sparsity produced by the Lottery Ticket Hypothesis work). This technique iteratively trains a network, prunes it by simply removing the 20% of weights with the lowest magnitude throughout the network, rewinds the weights to their values early in training, then re-trains and repeats. Using this technique results in sparse networks that reach the same accuracy as the original dense network: we can prune a ResNet-20 to 21% density and a VGG-16 to only 1% density without any loss in accuracy. However, these sparse networks are not uniformly sparse across all layers: early layers (with few channels, and therefore few parameters) tend to be minimally pruned and end up dense. However later layers (with many channels and are correspondingly larger) tend to be pruned to be sparser. The layerwise sparsity rates for ResNet-20 and VGG-16 are presented in Table 1.

Sparse Convolution with CSR The following code shows the algorithm that we use to implement convolutions that exploit weight sparsity [28]. We store the weight tensors in a CSR-like format (Compressed Sparse Row). This format is created as follows: first, we flatten the original weight tensor which has the following dimensions (OutputFeatures, InputFeatures, K, K) [2] to (OutputFeatures, InputFeatures×K×K); then we compress the rows of the resulting matrix using CSR.

```
for each output channel n
  for j in (W.rowptr[n], W.rowptr[n+1]) {
    off = W.colidx[j]; coeff = W.value[j];
    for (int y = 0; y < H_OUT; ++y)
      for (int x = 0; x < W_OUT; ++x)
        out[n][y][x] += coeff*in[y*W_OUT+x+off)]
  }
```

[2]k is the size of the convolution filter (e.g., 3×3)

4 Expressing and Optimizing Recurrent Neural Networks

Many state-of-the-art DNN compilers do not allow users to express dynamic RNNs. Halide [29], for example, is designed to express programs with acyclic dependence graphs (which excludes dynamic RNNs); this restriction is imposed by the Halide language and compiler to guarantee the correctness of optimizations. To avoid this overconservative language restriction, TIRAMISU relies on dependence analysis instead to check for the correctness of code transformations, enabling the user to express dynamic RNNs and optimize them.

In order to parallelize the execution of multilayer-LSTMs, TIRAMISU applies a transformation known as iteration space skewing which exposes wavefront parallelism hidden in multilayer-LSTMs. Such parallelization is necessary for increasing GPU occupancy when targeting GPUs, it is also necessary to parallelize multilayer-LSTMs when targeting distributed architectures.

5 Evaluation

We evaluate TIRAMISU on a set of deep learning benchmarks. We compare it with the Intel MKL-DNN (1.0) and cuDNN (7.0) libraries which provide highly optimized implementations for Intel multicore CPUs and Nvidia GPUs.

The CPU evaluation is performed on an 8-core Intel i7-6700HQ CPU, 16 GB RAM, Ubuntu 18.04. The GPU evaluation is performed on an Nvidia Pascal P4 GPU. Each experiment is repeated 30× and the median time is reported.

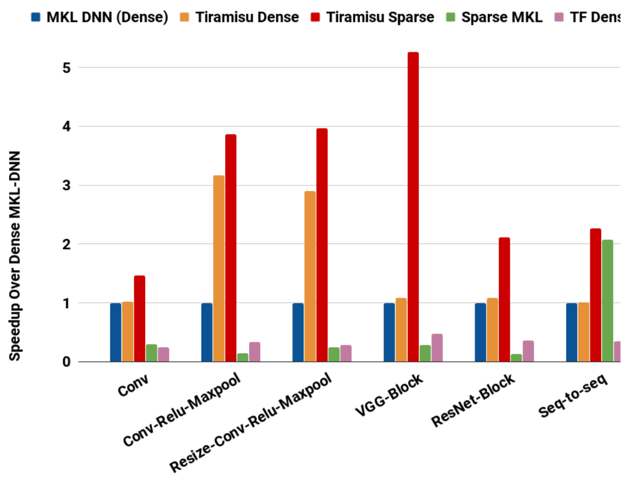

Figure 1: Speedups over Intel MKL-DNN.

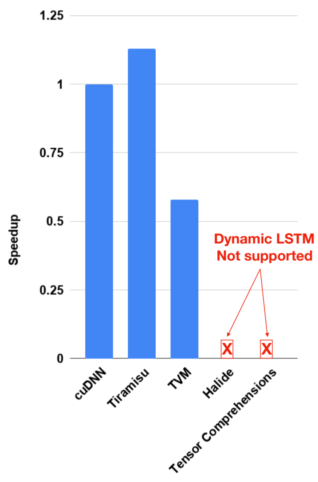

Figure 2: Speedups over cuDNN (dense).

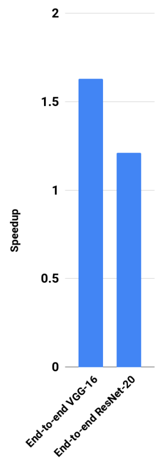

Figure 3: Speedups over MKL-DNN.

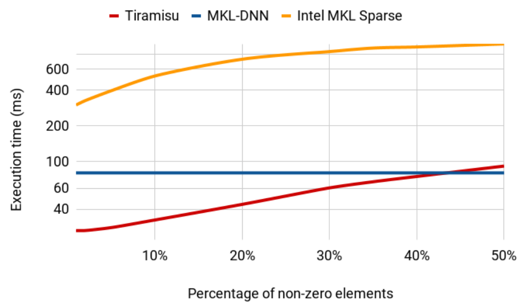

Figure 4: Break-even point for sparse convolution.

Feature	Tiramisu	TC	TVM	Halide
Express dynamic RNNs	Yes	No	Yes	No
Optimize dynamic RNNs	Yes	No	No	No
Express/optimize sparse DNNs	Yes	No	No	No
Generate distributed Code	Yes	No	No	Yes
Scheduling language	Yes	No	Yes	Yes
Support all affine transformations	Yes	Yes	No	No

Figure 5: Comparison of DNN compilers

The deep learning benchmarks include `Conv` (a direct implementation of a neural network convolution layer), `Conv-Relu-Maxpool` (a block of three layers, a direct convolution followed by a rectified linear unit followed by maxpooling), `Resize-Conv-Relu-Maxpool` (the same benchmark as the previous one but preceded by an image resizing step for preprocessing), `VGG` (a block of the VGG neural network [30]), `ResNet` (a block of the ResNet neural network [19]), and `Seq-to-seq` (a multilayer-LSTM that translates a sequence to another sequence [31]).

The use of a sparse convolution is not always profitable. Above certain density levels, a dense convolution implementation is more profitable than the sparse counterpart due to the overhead that the sparse implementation adds. Figure 4 shows the break-even density level (43.5%) after which a dense convolution implementation is faster than its sparse counterpart. The Intel MKL sparse implementation relies on sparse matrix multiplication and is slower than both implementations mainly due to the extra cost of lowering [28].

The `VGG-Block` and `ResNet-Block` benchmarks in Figure 1 are two representative blocks from the `VGG` [30] and `ResNet` [19] neural network architectures (a block is a repetitive sequence of layers in the neural network). We use the same sizes and parameters as in the original architectures. The sparse weights are obtained by applying the LTH pruning technique [11]. The blocks are chosen to be representative: first we exclude all the blocks that have a density level above 43.5% and which should have a dense implementation; then, we compute the median of the weight density of the the remaining blocks; the chosen blocks have a density that is the closest to the median density. Based on this methodology, we find that block 10 in both `ResNet` and `VGG` has the median density level (as shown in Table 1). The density level for block 10 is 16.1% in `ResNet` and 1% in `VGG`. For `seq-to-seq`, we use the same architecture and sizes used in [39] (4 LSTM layers, 100 elements in the input sequence and 1024 hidden parameters), and use 15% as a uniformly distributed density level [21].

Figure 1 shows a comparison between the performance of code generated by TIRAMISU (multicore CPU) and reference DNN libraries and frameworks. The baseline is the Intel MKL-DNN library (dense). The comparison includes the TIRAMISU implementation for dense weights, the TIRAMISU implementation for sparse weights, an implementation using Intel MKL sparse and the TensorFlow framework.

TIRAMISU outperforms the highly optimized Intel MKL-DNN library by up to $3\times$ in `Conv-Relu-Maxpool` and `Resize-Conv-Relu-Maxpool` due to operator fusion. TIRAMISU fuses the operators `Conv`, `Relu`, Maxpool (and `resize`) whereas Intel MKL-DNN has an implementation where only `Conv` and `Relu` are fused. For the sparse implementation, TIRAMISU outperforms the Intel MKL-DNN implementation by up to $5\times$. In `Conv-Relu-Maxpool` and `Resize-Conv-Relu-Maxpool`, in addition to the sparse implementation, we apply operator fusion. Figure 3 shows end-to-end speedups for sparse TIRAMISU compared to MKL-DNN (dense).

LSTM Optimization on GPU Figure 2 compares the TIRAMISU GPU implementation of the `seq-to-seq` neural network with that of the cuDNN library [26], TVM, Halide and Tensor Comprehensions. While TIRAMISU and cuDNN use iteration space skewing to parallelize the multi-layer LSTM and increase the occupancy of the GPU, the TVM implementation does not support iteration space skewing and thus suffers from lower GPU occupancy. Halide and Tensor Comprehensions do not support dynamic LSTMs. In addition to the use of iteration space skewing to parallelize the `seq-to-seq` benchmark, the TIRAMISU implementation fuses multiple matrix multiplications into fewer multiplications to increase the GPU occupancy and uses the CUDA streams API to achieve concurrency on multiple GPUs [2]. TIRAMISU is faster than cuDNN in particular, because TIRAMISU tunes the number of fused matrix multiplications while knowing the size of the matrix multiplication whereas cuDNN does not provide such capability. In a separate experiment, we have found that the optimal number of fused matrix multiplication depends on the size of the LSTM matrix multiplication operations therefore.

6 Related Work

Tensor Comprehensions [33] and Diesel [10] are fully automatic polyhedral compilers for deep learning designed mainly to target GPUs. Unlike Tensor Comprehensions and Diesel, TIRAMISU has a scheduling language and therefore allows the user to have fine grain control over optimizations. TVM [38] is another DNN compiler designed for targeting multiple hardware architecture. It has a scheduling language and uses machine-learning-based auto-tuning. TVM is not polyhedral though. It uses intervals to represent loop bounds and loop transformations which prevents TVM from applying certain transformations such as iteration space skewing (which is necessary for optimizing RNNs such as multilayer-LSTMs and increase GPU occupancy). Other machine learning domain specific compilers include TensorFlow XLA [1], DLVM [35], Latte [32] and SWIRL [34]. Among all of the previous compilers, TIRAMISU is the only compiler that supports sparse DNNs. Figure 5 shows a comparison with some of these compilers (`TC` in the table stands for Tensor Comprehensions).

Polyhedral compilers such as PENCIL [5, 4], Pluto [7], Polly [15], and PolyMage [25] are fully automatic. While such fully automatic compilers provide productivity, they may not always obtain the best performance. This is due to many reasons: these compilers do not implement some key optimizations such as array packing [14], register blocking, data prefetching (which are all supported by TIRAMISU). Besides, they do not have a precise cost-model to decide which optimizations are profitable. For example, the Pluto [7] automatic scheduling algorithm (which is used for automatic scheduling in Pluto, PENCIL, Polly, and Tensor Comprehensions) tries to minimize the distance between producer and consumer statements while maximizing outermost parallelism, but it does not consider the data layout, redundant computations, or the complexity of the control of the generated code. Instead of fully automatic scheduling, TIRAMISU uses a more pragmatic approach and relies on a set of scheduling commands, giving the user full control over scheduling.

Other polyhedral compilers such as AlphaZ [37], CHiLL [8, 16], URUK [13], and Transformation Recipes [17] allow users to express high-level transformations using scheduling commands. Since these frameworks are polyhedral, they can express any affine transformation. Their scheduling languages though only implement a subset of the transformations that are necessary to get peak performance. For example, they do not implement optimizations such as array packing, prefetching and register blocking.

Halide [29] is an image processing DSL that has a scheduling language; however, it uses intervals to represent iteration spaces instead of the polyhedral model. This limits the expressiveness of Halide. For example, unlike TIRAMISU, Halide cannot naturally represent non-rectangular iteration spaces. It also cannot perform many complex affine transformations, such as iteration space skewing which is necessary for optimizing RNNs. In addition, Halide assumes that the program has an acyclic dataflow graph in order to simplify checking the legality of a schedule. This prevents users from expressing many programs with cyclic dataflow; for example, Halide does not allow the fusion of two loops (using the `compute_with` command) if the second loop reads a value produced by the first loop. While this rule avoids illegal fusion, it prevents fusing many legal common cases. TIRAMISU avoids over-conservative constraints by relying on dependence analysis to check for the correctness of code transformations, enabling more possible schedules.

Exploiting sparsity in deep neural networks has been the subject of multiple projects. Park et al. [28] presented a fast algorithm for implementing sparse direct convolutions (on which we based our implementation), whereas Xuhao Chen [9] Parashar et al. [27] on the other hand presented a hardware accelerator for sparse CNNs.

Acorns [36] is a framework designed mainly to optimize DNNs with input sparsity. It has a set of template codes for neural network operators and does not implement advanced loop nest optimizations such as iteration space skewing. Acorns introduces a data layout that exploits the structure of sparsity of input data in certain domains (LiDAR, face detection, character recognition, ...) where only certain specific regions of the input are non-zero. Unlike Acorns, TIRAMISU focuses on sparsity in weights.

7 Conclusion

In this paper, we demonstrate a DNN compiler that has two unique features: (1) it can generate efficient code for sparse DNNs; (2) it can optimize dynamic RNNs. TIRAMISU can apply complex loop transformations thanks to the use of the polyhedral representation; and it relies on the use of scheduling commands, therefore it allows fine control over which optimizations to apply which allows TIRAMISU to reach high performance. We evaluate TIRAMISU by implementing a set of deep learning benchmarks and show that TIRAMISU matches and outperforms the Intel MKL-DNN and cuDNN libraries by up to $5\times$ and outperforms state-of-the-art compilers by up to $2\times$.

References

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[2] A. Akkas. Efficient memory and gpu operations for tiramisu compiler. S.m. thesis, Massachusetts Institute of Technology, Cambridge, MA, May 2019.

[3] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe. Opentuner: An extensible framework for program autotuning. In *International Conference on Parallel Architectures and Compilation Techniques*, Edmonton, Canada, August 2014.

[4] R. Baghdadi, U. Beaugnon, A. Cohen, T. Grosser, M. Kruse, C. Reddy, S. Verdoolaege, A. Betts, A. F. Donaldson, J. Ketema, J. Absar, S. v. Haastregt, A. Kravets, A. Lokhmotov, R. David, and E. Hajiyev. Pencil: A platform-neutral compute intermediate language for accelerator programming. In *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT)*, PACT '15, pages 138–149, Washington, DC, USA, 2015. IEEE Computer Society.

[5] R. Baghdadi, A. Cohen, T. Grosser, S. Verdoolaege, A. Lokhmotov, J. Absar, S. van Haastregt, A. Kravets, and A. F. Donaldson. PENCIL language specification. Research Rep. RR-8706, INRIA, 2015.

[6] R. Baghdadi, J. Ray, M. B. Romdhane, E. Del Sozzo, A. Akkas, Y. Zhang, P. Suriana, S. Kamil, and S. Amarasinghe. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO 2019, pages 193–205, Piscataway, NJ, USA, 2019. IEEE Press.

[7] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *PLDI*, pages 101–113, 2008.

[8] C. Chen, J. Chame, and M. Hall. Chill: A framework for composing high-level loop transformations. Technical Report 08-897, U. of Southern California, 2008.

[9] X. Chen. Escort: Efficient sparse convolutional neural networks on gpus. *CoRR*, abs/1802.10280, 2018.

[10] V. Elango, N. Rubin, M. Ravishankar, H. Sandanagobalane, and V. Grover. Diesel: Dsl for linear algebra and neural net computations on gpus. In *Proceedings of the 2Nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2018, pages 42–51, New York, NY, USA, 2018. ACM.

[11] J. Frankle, G. Karolina Dziugaite, D. M. Roy, and M. Carbin. Stabilizing the Lottery Ticket Hypothesis. In *arXiv*, page arXiv:1903.01611, 2019.

[12] T. Gale, E. Elsen, and S. Hooker. The State of Sparsity in Deep Neural Networks. *arXiv*, page arXiv:1902.09574, Feb 2019.

[13] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *International Journal of Parallel Programming*, 34(3):261–317, 2006.

[14] K. Goto and R. A. v. d. Geijn. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw.*, 34(3):12:1–12:25, May 2008.

[15] T. Grosser, A. Groslinger, and C. Lengauer. Polly - performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(4), 2012.

[16] M. Hall, J. Chame, C. Chen, J. Shin, G. Rudy, and M. M. Khan. *Loop Transformation Recipes for Code Generation and Auto-Tuning*, pages 50–64. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[17] M. Hall, J. Chame, C. Chen, J. Shin, G. Rudy, and M. M. Khan. Loop transformation recipes for code generation and auto-tuning. In *Proceedings of the 22Nd International Conference on Languages and Compilers for Parallel Computing*, LCPC'09, pages 50–64, Berlin, Heidelberg, 2010. Springer-Verlag.

[18] S. Han, J. Pool, J. Tran, and W. Dally. Learning both weights and connections for efficient neural network. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 1135–1143. Curran Associates, Inc., 2015.

[19] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[20] Y. He, J. Lin, Z. Liu, H. Wang, L.-J. Li, and S. Han. Amc: Automl for model compression and acceleration on mobile devices. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 784–800, 2018.

[21] N. Kalchbrenner, E. Elsen, K. Simonyan, S. Noury, N. Casagrande, E. Lockhart, F. Stimberg, A. v. d. Oord, S. Dieleman, and K. Kavukcuoglu. Efficient neural audio synthesis. *arXiv preprint arXiv:1802.08435*, 2018.

[22] Y. Le Cun, J. S. Denker, and S. A. Solla. Optimal brain damage. In *Advances in Neural Information Processing Systems*, pages 598–605. Morgan Kaufmann, 1990.

[23] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf. Pruning filters for efficient convnets. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*, 2017.

[24] S. G. Michael H. Zhu. To prune, or not to prune: Exploring the efficacy of pruning for model compression, 2018.

[25] R. T. Mullapudi, V. Vasista, and U. Bondhugula. Polymage: Automatic optimization for image processing pipelines. *SIGARCH Comput. Archit. News*, 43(1):429–443, Mar. 2015.

[26] Nvidia. *cuDNN Library User Guide*, 2017.

[27] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. S. Emer, S. W. Keckler, and W. J. Dally. SCNN: an accelerator for compressed-sparse convolutional neural networks. *CoRR*, abs/1708.04485, 2017.

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

[29] J. Ragan-Kelley, A. Adams, S. Paris, M. Levoy, S. Amarasinghe, and F. Durand. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Trans. Graph.*, 31(4):32:1–32:12, July 2012.

[30] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[31] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. *CoRR*, abs/1409.3215, 2014.

[32] L. Truong, R. Barik, E. Totoni, H. Liu, C. Markley, A. Fox, and T. Shpeisman. Latte: A language, compiler, and runtime for elegant and efficient deep neural networks. *SIGPLAN Not.*, 51(6):209–223, June 2016.

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

[34] A. Venkat, T. Rusira, R. Barik, M. Hall, and L. Truong. Swirl: High-performance many-core cpu code generation for deep neural networks. *The International Journal of High Performance Computing Applications*, 0(0):1094342019866247, 0.

[35] R. Wei, V. S. Adve, and L. Schwartz. DLVM: A modern compiler infrastructure for deep learning systems. *CoRR*, abs/1711.03016, 2017.

[36] X. F. Xiao Dong, Lei Liu. Acorns: A framework for accelerating deep neural networks with input sparsity. In *Proceedings of the 2019 International Conference on Parallel Architecture and Compilation (PACT)*, PACT '19, Seattle, WA, USA, 2019. IEEE Computer Society.

[37] T. Yuki, G. Gupta, D. Kim, T. Pathan, and S. Rajopadhye. Alphaz: A system for design space exploration in the polyhedral model. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 17–31. Springer, 2012.

[38] C. Zhang, P. Patras, and H. Haddadi. Deep learning in mobile and wireless networking: A survey. *IEEE Communications Surveys & Tutorials*, 2019.

[39] M. Zhang, S. Rajbhandari, W. Wang, and Y. He. Deepcpu: Serving rnn-based deep learning models 10x faster. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 951–965, Boston, MA, July 2018. USENIX Association.