Branch Prediction with Multi-Layer Neural Networks: The Value of Specialization

Siavash Zangeneh*
siavash.zangeneh@utexas.edu

Stephen Pruett*
stephen.pruett@utexas.edu

Yale Patt*
patt@ece.utexas.edu

**The University of Texas at Austin*

***Abstract*—Multi-layer neural networks show promise in improving branch prediction accuracy. Tarsa et al. have shown that convolutional neural networks (CNNs) can accurately predict many branches that state-of-the-art branch predictors cannot. Yet, strict latency and storage constraints make naive adoption of typical neural network architectures impractical. Thus, it is necessary to understand the unique characteristics of branch prediction to design constraint-aware neural networks. This paper studies why CNNs are so effective for two hard-to-predict branches from the SPEC benchmark suite. We identify custom prediction algorithms for these branches that are more accurate and cost-efficient than CNNs. Finally, we discuss why out-of-the-box machine learning techniques do not find optimal solutions and propose research directions aimed at solving these inefficiencies.**

I. Introduction

Branch prediction is a success story of using machine learning to improve computer architecture. Predictors like TAGE and Perceptron use learning algorithms to capture correlations between branch history and future branch outcomes. Unfortunately, TAGE and Perceptron are not capable of isolating correlated branches but rely on brute force to capture it [12]. While this approach is effective for many branches, there remain a small number of hard-to-predict static branches that disproportionately drive up mispredictions per kilo-instruction (MPKI) [8]. These hard-to-predict branches are plagued by variations in the long branch history register [1], which generate a large number of signatures for runtime predictors to memorize.

Tarsa et al. [14] introduce a Convolutional Neural Network (CNN) that directly isolates correlated branches in the long branch history. The main insight of Tarsa's work is that training the CNN, widely considered to be too expensive (compute and latency) to do on chip at runtime, can be done offline to detect invariant correlations between the same branches in a program. Once the heavy lifting of detecting correlations is done, a relatively simple online inference engine is used to generate predictions at runtime. While we believe offline training is groundbreaking for branch prediction, the implementation introduced by Tarsa does not live up to its potential. Specifically, Tarsa's runtime predictor requires an extra 148.6 KB of storage, increasing the branch predictor budget by 232%, while only improving the MPKI of branch-sensitive benchmarks by only 3.9%.

[1]Some configurations of TAGE-SC-L use up to 1000 bits of history

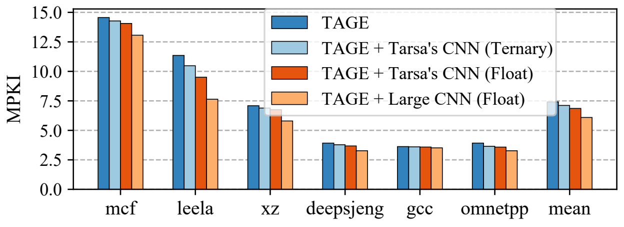

Fig. 1: MPKI of branch-sensitive SPEC2017 benchmarks.

We argue the costliness of Tarsa's predictor stems from the fact that the CNN architecture was not customized for branch prediction. This paper examines case studies of real-world branches where CNNs significantly outperform traditional predictors. We use those examples to identify the benefits and inefficiencies of CNN branch predictors. Finally, we highlight new areas of research that we believe would improve the overall cost of CNNs for branch prediction.

II. Background

The use of multi-layer neural networks represents a new chapter for branch prediction. This class of predictors is capable of directly isolating correlated branches in the branch history, discovering more sophisticated forms of correlation than was previously detectable by online predictors such as TAGE [6], [9], [10], [13] and Perceptron [5], [7]. The key insight enabling this breakthrough is that neural networks can be trained offline, thus avoiding the use of highly expensive and impractical training algorithms at runtime. While the networks we discuss in this paper may seem small in the world of deep learning, training the networks requires cascading long-latency floating-point operations, which is far beyond any reasonable budget for the branch predictor, even for a high-performance core. Offline training, however, allows the network to be trained once, after compilation, and the weights to be reused until the program is compiled again.

Offline training discovers correlated branches by training the network on many different runs of the same program, but with different inputs. Each input exercises a different control flow path in the program, exposing new combinations of branches. Once the network has been trained on enough examples, it learns which branches are correlated and which are not. This information (i.e., the trained weights) can be applied to the branch history at runtime to efficiently and accurately predict the result of hard-to-predict branches.

```
1  void qsort(void* a, size_t n, ...) {
2      if (n < 7) {
3          insertion_sort(a, n, ...);
4          return;
5      }
6
7      // start partitioning
8      for (int i = 0 to n) {
9          if (a[i] < pivot) {
10             ... // insert a[i] in the left partition
11         }
12         if (a[i] > pivot) {
13             ... // insert a[i] in the right partition
14         }
15     }
16     // finished partitioning
17
18     if (size of the left partition > 1) {
19         qsort(...) // recurse on the left partition
20     }
21     if (size of the right partition > 1) {
22         qsort(...) // recurse on the right partition
23     }
24 }
```

Listing 1: Simplied pseudo-code of *qsort*.

While Tarsa's work addressed the bulk of the cost by training the CNN offline, their online inference engine is still costly when compared to other online branch predictors. The inference engine is a 2 layer network, consisting of a 32-filter convolution layer and a fully connected layer. The predictor uses a history length of 200, with each history element consisting of 7 bits of PC and 1 bit of direction. The history is converted to 1-hot format, producing a 200x256 bit input for the CNN to process. The resulting CNN is a 15 stage circuit (we estimate 4 cycles of latency) that requires 5.125 KB of storage per branch and improves MPKI by 3.9%. This increases the total storage allocated to the predictor by 232%, which may be unacceptable to chip designers. Moreover, as Fig. 1 shows, the MPKI reduction of Tarsa's CNN is only a fraction of what is achievable by a larger and more capable CNN (MPKI reduction of Large CNN is 18.5%).

We argue that the cost of Tarsa's online inference engine could be reduced if the network and training algorithms were customized specifically for branch prediction. Improving the cost-effectiveness of CNNs also has the additional benefit of enabling more capable (e.g. more layers, more filters) neural networks, resulting in more accuracy. In the next section, we discuss two branches in the *qsort* algorithm that benefit from deep neural networks. We discuss the properties of those branches that result in a poor accuracy on TAGE, and the shortcomings of Tarsa's predictor that cause its resources to be used inefficiently.

III. Case Studies

We use two hard-to-predict branches from the *qsort*[2] function as examples of branches that a CNN can accurately predict. We choose *qsort* because it is hot code in *mcf*, one of SPEC 2017 Integer benchmarks [2], and because it contains several branches that have sophisticated correlations with the branch history. Listing 1 shows a simplified pseudo-code[3] for SPEC's implementation of *qsort* with the two example hard-to-predict branches highlighted in yellow. Both example branches can be predicted 100% accurately; however, both are predicted poorly by TAGE[4]. For each example, we present a custom predictor that predicts the branch perfectly. Then, we show why a CNN can predict the branch much more accurately than TAGE. Finally, we identify inefficiencies in Tarsa's CNN by contrasting it to our targeted solution, which predicts with higher accuracy, lower latency, and better storage-efficiency.

[2]*qsort* is a C library function for in-place sorting, typically implemented using the Quicksort algorithm.

```
int left_partition_size;

void update(next_execution_line) {
   switch(next_execution_line) {
      case qsort::line7: left_partition_size = 0;
      case qsort::line10: left_partition_size += 1;
   }
}

bool pred() {
   return (left_partition_size < 2);
}
```

Listing 2: Perfect custom predictor for the if-statement in line 18.

A. Case Study 1: branch guarding the left recursion (line 18)

After the partitioning phase of *qsort* is done (lines 7-16), the algorithm decides whether to recursively call *qsort* on each partition (lines 19 and 22). In this case study, we focus on the branch in line 18 of Listing 1. We assume that for each if-statement, the taken direction of the guard branch skips the body. TAGE-SC-L predicts this branch with 94.7% accuracy, which is only slightly better than the static bias of the branch (92.7% not-taken). However, we know the branch will be not-taken (i.e., *qsort* will be recursively called) only if the partition has at least two items in it. Therefore, the size of the partition can be used to predict the branch with 100% accuracy — predict not-taken only if the partitioning phase has inserted at least two items in the left partition (line 10 was executed at least 2 times). We can use the branch history to determine the number of times line 10 was executed. If the count reaches two, then we know there are at least two elements in the left partition and the branch at line 18 should be predicted not-taken (or taken if the count is less than two). Note, there are two caveats to this approach. First, the predictor must know when the partitioning phase begins so that it can initialize the count to zero. Second, once compiled, the resulting assembly code for Listing 1 contains 3 branches (instead of just one) that guard the execution of line 10. Therefore, the branch predictor should directly isolate those three branches to count the number of elements inserted into the left partition.

Listing 2 defines the update and prediction algorithm for a targeted branch predictor that implements a perfect prediction strategy for this case study. It consists of three components: the predictor state (left_partition_size), an update algorithm

[3]The Listing does not show all of the implementation details required for high performance, but does capture the branch behavior as it pertains to the two example branches.

[4]We use the hot code of *mcf* for evaluating prediction accuracy of the example branches throughout the paper.

that updates the state every time a branch is fetched, and a prediction function. To implement this algorithm in hardware, we require a 2-bit saturating counter for left_partition_size, a single register to track the PC of the branch leading to qsort::line7, and 3 registers used to track the PCs of branches that guard qsort::line10, amounting to 198 bits of storage.

Even though this simple prediction algorithm exists, TAGE cannot predict this branch accurately because it cannot distinguish the correlated branches in the history (line 9) from the uncorrelated ones (all other branches). As a result, the pollution in the branch history creates too many signatures for TAGE to memorize, resulting in a low accuracy.

Tarsa's CNN predictor, however, uses a convolution layer that acts as a filter for identifying correlated branches and removes uncorrelated branches. Once the uncorrelated branches have been removed, the final layer of the network can easily check the size of the partition by counting the not-taken occurrences of line 9. In fact, both the ternary and the floating-point version of Tarsa's CNN predict this branch with high accuracy (99.35% and 99.7% respectively). Note, however, that we only want to calculate the size of the most recent partition in the history. Our targeted solution handled this by resetting the counters before the partitioning began. Tarsa's CNN, however, must learn on its own which regions of the history register are important. This leads to an inefficiency between Tarsa and the custom logic that will be discussed later.

Unfortunately, the high prediction accuracy of Tarsa's CNN comes at a high storage cost. Table I compares the accuracy and storage of TAGE [5], Tarsa's CNN predictors, and custom logic [6]. Even though Tarsa's CNN is much more accurate than TAGE, it is still not perfect and requires unnecessarily large storage.

TABLE I: case study 1: accuracy and storage of predictors.

	TAGE	Tarsa Ternary	Tarsa Float	Large CNN	Custom Logic
Accuracy	94.7%	99.35%	99.7%	$\sim$100.0%	100%
Size	N/A	5.1 KB	82 KB	17.7 MB	198 bits

Why does Tarsa's CNN not reach 100%? Identifying the most immediate partitioning phase in the global branch history is a highly non-linear task. Thus, Tarsa's CNN with only one fully-connected layer cannot ever learn to predict this branch the optimal way. The CNN compensates for this inability by learning to use any correlated branch in the history to improve its accuracy. For example, if the sizes of left and right partitions are correlated in the training set, the CNN will use the right partition size for prediction. In general, this overfitting may lower prediction accuracy at runtime, but in this case, such a data-driven approach is sufficient for $> 99\%$ accuracy.

Another disadvantage of a CNN compared to the optimal algorithm is the relatively small history length. We measured the minimum history length to identify at least two items have been inserted into the left partition in our test set. The median distance is 30 branches, the 99th percentile is 156, and the maximum distance is 710. Since the history length of Tarsa's CNN is 200 branches, it cannot accurately determine the correct size of the partitions. In this case, however, a 200-branch history happens to be enough for determining whether the partition has more than one element or not, which is all that is needed for correct prediction. In general, predictors that seek to achieve high accuracy need to worry about what history lengths are needed to cover most of the cases.

Sources of storage-inefficiency As described earlier, Tarsa's CNN model is incapable of learning the optimal prediction algorithm and instead relies on using any correlated branches in the history. Thus, it needs many convolution filters to identify all useful correlated branches. Moreover, some degree of over-parameterization is necessary for convergence when training multi-layer neural networks [1], which by definition implies sub-optimal predictor size. This factor can be somewhat alleviated with post-training network pruning.

Another source of storage-inefficiency is the fully-connected layer, which is a much more general function than needed. Note that the inference engine not only needs to store all the fully-connected weights but also should buffer the convolution outputs that feed the fully-connected layer. In our custom design, this was replaced by incrementing a 2-bit saturating counter at the appropriate times.

B. *Case Study 2: branch guarding insertion sort (line 2)*

A common technique for speeding up quick-sort is to switch sorting algorithms once a partition is smaller than some threshold. Line 2 in Listing 1 is the if-statement that controls this switch. This branch is very hard-to-predict for TAGE with 66.4% accuracy (the static bias of the branch is 56.2% taken).

At first glance, this branch may seem as predictable as the branch in case study 1. For all recursive calls to *qsort*, the value of *n* is produced by the partitioning phase in the caller instance of *qsort*. Thus, similar to case study 1, a branch predictor can determine *n* by tracking insertions into the left and right partitions. However, this prediction task is actually much more difficult than the prediction task in case study 1 because the caller instance of *qsort* may appear arbitrarily deep into the branch history. A single fully-connected layer, especially if ternarized, is too simple to learn this behavior. As a result, the accuracy of the ternary and the floating-point version of Tarsa's CNN are 82.4% and 88.4% respectively.

The poor accuracy of Tarsa's CNN does not mean that CNNs cannot predict this branch accurately. We train a larger CNN (e.g. more filters, wider convolutions, longer history, 2 fully-connected layers) to predict this branch. The large CNN predicts this branch with 98.2% accuracy, albeit as the cost of a 17.7MB model. Of course, such a large prediction model is not helpful at runtime. The more interesting question is whether a CNN can be accurate and cost-efficient at the same time.

[5]Storage cost of TAGE per branch is dynamic and depends on the allocation pressure from other branches, so we will not quantify it.

[6]Custom Predictor defined in Listing 2.

```
1  int left_partition_size, right_partition_size;
2  bool left_recursion;
3  stack<int> right_partition_predictions;
4
5  void update(next_execution_line) {
6     switch(next_execution_line) {
7        // Determining partition sizes
8        case qsort::line7:
9           left_partition_size = 0;
10          right_partition_size = 0;
11       case qsort::line10: left_partition_size += 1;
12       case qsort::line13: right_partition_size += 1;
13
14       // Push the prediction for the right
15       // partition in a stack
16       case qsort::line16:
17          if right_partition_size > 1:
18             pred = (right_partition_size > 6);
19             right_partition_predictions.push(pred);
20
21       // Update left_recursion before each call
22       case: qsort::line19: left_recursion = true;
23       case qsort::line22: left_recursion = false;
24    }
25 }
26
27 bool pred() {
28    if left_recursion:
29       return (left_partition_size > 6);
30    else:
31       return right_stack.pop();
32 }
```

Listing 3: Perfect custom predictor for the if-statement in line 2.

Listing 3 shows an accurate and cost-efficient algorithm for predicting this branch by observing the incoming branch stream. First, similar to Listing 2, it tracks insertions into the left and right partitions using saturating counters (lines 7-12). After the partitioning, the algorithm produces a prediction for the if-statement in qsort::line2 of the right partition and pushes the prediction in a prediction stack (lines 16-19). Furthermore, before a recursive call on either the left or right partition, it sets the flag *left_recursion* accordingly (lines 22-23). Finally, to make a prediction, if the *left_recursion* flag is set, it simply uses the size of the left partition to make a prediction, otherwise, it pops a prediction off the stack. This algorithm fails when predicting the root of the recursion tree, but is otherwise completely accurate. To implement this algorithm in hardware, we require 11 registers to track the PC, branch-direction pairs, two 3-bit counters for determining the left and right partition sizes, and a 64-entry stack (1-bit per entry) used to hold the predictions for the right partition, amounting to a total of 609 bits.

Similar to the previous case study, representing this optimal algorithm by a CNN is unrealistic because of insufficient history length. This case study is even more difficult because if *qsort* is entered because of a recursive call on the right partition, the global branch history is polluted with an unknown number of partitioning branches because of the earlier recursion on the left partition. For predicting 90% of the instances of this branch in *qsort*, the optimal algorithm discards up to 641 youngest branches in the global history.

TABLE II: case study 2: accuracy and storage of predictors.

	TAGE	Tarsa Ternary	Tarsa Float	Large CNN	Custom Logic
Accuracy	66.4%	82.4%	88.4%	98.2%	100%
Size	N/A	5.1 KB	82 KB	17.7 MB	609 bits

As Table II shows, there remains a large gap between Tarsa's CNN and the accuracy of a large CNN or the cost-efficiency of a targeted solution. To bridge this gap, it is necessary to specialize CNNs for the task of branch prediction. In the next section, we describe several research directions we believe will lead to efficient CNN branch predictors.

IV. Research Directions

In this section we discuss our view of important research directions to continue to improve the prediction accuracy and storage efficiency of multi-layer neural networks for branch prediction.

Long history lengths. Case study 2 showed that some branches benefit from longer history lengths, but naively increasing the history length is impractical because of inference hardware constraints. Traditional online predictors [11], [13] use geometrically increasing history registers to conserve storage by using short history lengths for most branches and long history lengths only when required.

Specialized structures. In both case studies, CNNs are worse than custom predictors in terms of both accuracy and storage-efficiency. This is because the CNN must learn functions that the custom logic was directly programmed to perform. On the other hand, the custom predictors are not general enough to deploy in an actual branch predictor. What we need is a predictor that is general enough to learn to predict branches in new algorithms, but contains enough custom logic that it does not need to re-learn functions that are common among many branches. If we implement custom structures targeted towards common operations, we can bridge the gap between a trainable predictor and our custom solutions. A key challenge will be finding ways to plug the custom logic into the network during training. Network designers may need to use functions that work with backpropagation, or use regularization, pruning, and/or post-training transformations to steer neurons towards the targeted custom hardware.

Input pre-processing. Branch history is an effective predictor for most branches. Unfortunately, branch history either loses or obfuscates a lot of semantic information from the program. The predictor is then forced to re-learn cryptic relationships that were known in advance by the compiler. Instead, we can take advantage of offline program analysis to pre-process the inputs to the CNN to simplify the prediction task. For example, let us reconsider case study 1. If the algorithm for identifying the relevant region in the history was produced through other means (e.g. static analysis), the role of the CNN would be to simply count the insertions into the partition up to a threshold. This task would only need 1 convolution layer and, in ideal training conditions, can be learned by a single fully-connected layer, resulting in a CNN with only a 0.21KB of storage with approximately 99.94% accuracy (the remaining inaccuracy is due to limited history length).

Hardware-aware training algorithms. As discussed throughout this paper, on-chip branch predictors have tight latency and storage constraints that must be obeyed. This makes quantization, pruning, and regularization very important. Tarsa et al. use the training algorithm of Courbariau et al. [4] to ternarize their CNN models. However, unlike the binarized neural networks studied by Courbariau et al., branch prediction accuracy significantly drops with quantization. This is partially because Tarsa's CNN is many orders of magnitude smaller than the CNN models that are evaluated in prior quantization work, increasing the likelihood of converging to bad local optimum solutions [1]. A more effective training strategy is to initially over-provision the network, then gradually regularize and prune the network to meet the hardware constraints. Such approaches are well studied in various prior work [3], [15], [16], albeit on larger models and more flexible inference engines. Hardware-aware training algorithms would allow us to use an over-parameterized network to assist with training, while still fitting in the predefined hardware budget for inference. For example, let us revisit case study 1. In theory a 1-filter CNN can predict the branch almost perfectly; however, training a 1-filter CNN results in only 93.0% accuracy, with a 2-filter CNN reaching 97.5%, and a 3-filter CNN reaching 99.7%. Now, if we over-parameterize the CNN with 4 filters, then use regularization to penalize redundant filters and prune the unused filters, we can achieve 99.7% accuracy with only 2 filters.

Recurrent Neural Networks. The most expensive component of a CNN branch predictor is the fully-connected layer. The fully connected layer is responsible for combining all of the signals extracted from the history by the CNN layer into a final prediction. To accomplish this, the hardware must buffer all signals produced by the CNN layer, producing buffers that require storage proportional to the length of the branch history register. An alternative approach would be to use Recurrent Neural Networks (RNNs). An ideal RNN branch predictor would process branches one at a time, updating its hidden state as branches are fetched. Sequential processing can simplify prediction tasks that rely on the order of branches in the history. For example, identifying the partitioning region of the case studies using RNNs is a relatively trivial task. However, there are other problems. The main hurdle is that there are no natural boundaries in the branch stream to indicate the start of input sequences. Using a fixed history length to form an input sequence does not work because the branch predictor does not know in advance when a hard-to-predict branch will be fetched. Still, even though we do not have a practical design for an inference engine, we have observed that small Gated Recurrent Units (GRUs) can also accurately predict many hard-to-predict branches, which leads us to believe that there is room for improvement using specialization and domain-specific techniques.

V. Conclusion

Convolutional neural networks are a helpful tool in improving the prediction accuracy of many hard-to-predict branches. However, hardware constraints for branch predictions make the naive adoption of CNNs impractical. This paper is a detailed study on the inefficiencies of a CNN compared to a perfect custom predictor. To address these inefficiencies, we argue for several research directions that share a common theme: specialization is necessary for enabling cost-efficient branch prediction with multi-layer neural networks.

References

[1] Z. Allen-Zhu, Y. Li, and Z. Song, "A convergence theory for deep learning via over-parameterization," *arXiv preprint arXiv:1811.03962*, 2018.

[2] J. Bucek, K.-D. Lange, and J. v. Kistowski, "Spec cpu2017: Next-generation compute benchmark," in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 41–42. [Online]. Available: https://doi.org/10.1145/3185768.3185771

[3] C. Chen, F. Tung, N. Vedula, and G. Mori, "Constraint-aware deep neural network compression," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018, pp. 400–415.

[4] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1," 2016.

[5] D. Jiménez, "Multiperspective perceptron predictor," in *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5)*, 2016.

[6] D. Jiménez, "Multiperspective perceptron predictor with tage," in *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5)*, 2016.

[7] D. A. Jimenez and C. Lin, "Dynamic branch prediction with perceptrons," in *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*, Jan 2001, pp. 197–206.

[8] C. Lin and S. J. Tarsa, "Branch prediction is not a solved problem: Measurements, opportunities, and future directions," *CoRR*, vol. abs/1906.08170, 2019. [Online]. Available: http://arxiv.org/abs/1906.08170

[9] P. Michaud, "An alternative tage-like conditional branch predictor," *ACM Trans. Archit. Code Optim.*, vol. 15, no. 3, pp. 30:1–30:23, Aug. 2018. [Online]. Available: http://doi.acm.org/10.1145/3226098

[10] S. Pruett, S. Zangeneh, A. Fakhrzadehgan, B. Lin, and Y. Patt, "Dynamically sizing the tage branch predictor," in *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5)*, 2016.

[11] A. Seznec, "Analysis of the o-geometric history length branch predictor," in *32nd International Symposium on Computer Architecture (ISCA'05)*, June 2005, pp. 394–405.

[12] A. Seznec, J. S. Miguel, and J. Albericio, "The inner most loop iteration counter: A new dimension in branch history," in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2015, pp. 347–357.

[13] A. Seznec, "TAGE-SC-L Branch Predictors," in *JILP - Championship Branch Prediction*, Minneapolis, United States, Jun. 2014. [Online]. Available: https://hal.inria.fr/hal-01086920

[14] S. J. Tarsa, C. Lin, G. Keskin, G. N. Chinya, and H. Wang, "Improving branch prediction by modeling global history with convolutional neural networks," *CoRR*, vol. abs/1906.09889, 2019. [Online]. Available: http://arxiv.org/abs/1906.09889

[15] K. Wang, Z. Liu, Y. Lin, J. Lin, and S. Han, "Haq: Hardware-aware automated quantization with mixed precision," 2018.

[16] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, "Learning structured sparsity in deep neural networks," in *Advances in neural information processing systems*, 2016, pp. 2074–2082.