# Lane Compression: A Lightweight Lossless Compression Method for Machine Learning on Embedded Systems

YOUSUN KO, ALEX CHADWICK, DANIEL BATES, and ROBERT MULLINS, University of Cambridge, United Kingdom

This paper presents Lane Compression, a lightweight lossless compression technique for machine learning which is based on a detailed study of the statistical properties of machine learning data. The proposed technique profiles machine learning data gathered ahead of run-time, and partitions values bit-wise into different *lanes* with more distinctive statistical characteristics. Then the most appropriate compression technique is chosen for each lane out of a small number of low-cost compression techniques. Lane Compression's compute and memory requirements are very low and yet it achieves a compression rate comparable to or better than Huffman coding. We evaluate and analyse Lane Compression on a wide range of machine learning networks for both inference and re-training. We also demonstrate the profiling prior to run-time and the ability to configure the hardware based on the profiling guarantee robust performance across different models and datasets. Hardware implementations are described and the scheme's simplicity makes it suitable for compressing both on-chip and off-chip traffic.

CCS Concepts: • **Theory of computation → Data compression**; • **Hardware → Reconfigurable logic and FPGAs**; • **Computer systems organization → Embedded systems**.

Additional Key Words and Phrases: machine learning, deep neural networks, ASIC

**ACM Reference Format:**
Yousun Ko, Alex Chadwick, Daniel Bates, and Robert Mullins. 2018. Lane Compression: A Lightweight Lossless Compression Method for Machine Learning on Embedded Systems. *ACM Trans. Embedd. Comput. Syst.* 37, 4, Article 111 (August 2018), 26 pages. https://doi.org/10.1145/1122445.1122456

## 1 INTRODUCTION

Machine learning techniques are rapidly being deployed across much of our computing infrastructure, from IoT and mobile devices to data centres. Consequently, demand for fast and efficient machine learning techniques has been increasing. Hardware accelerators can provide large gains over GPU and CPU-based solutions [16], but unfortunately gains can be limited by memory bandwidth and the overhead of off-chip memory access [25].

Data compression is one way to reduce the amount of memory traffic and improve the performance of workloads. Figure 1 shows a high-level view of a typical compression pipeline for machine learning. The figure shows that data in full-precision, usually 32-bit floating point, is compressed by a lossy transformation and/or lossless compression for training or inference. Lossy transformations compress data by reducing *the amount of information* in the data, and lossless compression compresses data by increasing *the density of information* in its representation. Lossy

Authors' address: Yousun Ko, yousun.ko@cl.cam.ac.uk; Alex Chadwick, alex.chadwick@cl.cam.ac.uk; Daniel Bates, daniel.bates@cl.cam.ac.uk; Robert Mullins, robert.mullins@cl.cam.ac.uk, University of Cambridge, 15 JJ Thomson Avenue, Cambridge, United Kingdom.


1539-9087/2018/8-ART111 $15.00
https://doi.org/10.1145/1122445.1122456

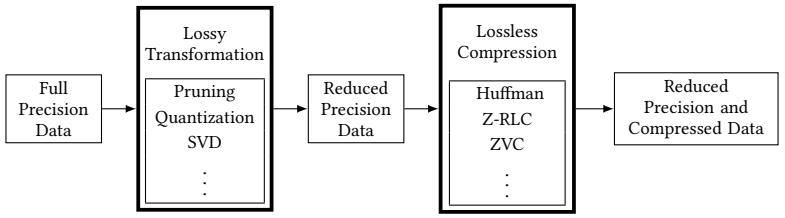



Fig. 1. Overview of compression in machine learning in a typical pipeline

transformation techniques for machine learning include pruning [19], singular value decomposition (SVD) [37], and quantization [26], and lossless compression techniques used for machine learning include Huffman coding [22], run-length coding on zeros (Z-RLC) [30], and zero-value compression (ZVC) [46].

Lossless compression for machine learning has been an active research field recently. Works has focused on compressing off-chip memory traffic [5, 6, 24, 31, 43, 46], data between computing units and cache [6, 8, 15, 17, 18, 52], or over the on-chip network traffic for a tiled architecture [6, 10, 43]. Lossless compression is also now employed in a number of commercial machine-learning accelerators [1, 36]. However, lossless compression has been considered as part of a compression pipeline rather than a stand-alone method in most previous work, hampering in-depth justification for the selected lossless compression methods and their algorithmic parameters. Such approaches also disrupt understanding the effectiveness of the selected method against a measure of the potential compressibility of the source data. We also note that the hardware complexities of previously suggested lossless compression techniques vary considerably, and some approaches are infeasible for implementation on resource-constrained hardware.

In this paper, we focus on lossless compression to learn how to use it for maximum effect for machine learning, and propose a new lightweight lossless compression technique for machine learning, called *Lane Compression*. Lane Compression demonstrates reliably high compression rates with a low hardware cost outperforming state-of-the-art lossless compression techniques for machine learning. To measure the effectiveness of the proposed method and previous lossless compression methods, we use the notion of *Shannon limit* which is the theoretical limit for symbol-based compression given by Shannon's marginal entropy [49]. Noting that lossy transformations and lossless compression are complementary, we show that combination of simple and low cost lossless compression and lossy transformation can lead to compression results competitive with highly aggressive and costly lossy transformation.

The contributions of this paper are as follows:

- We present key characteristics of the redundancy in machine learning data based on a detailed study of their statistical properties (see Section 4).
- We propose Lane Compression, a lossless compression technique for machine learning, that achieves compression rates comparable or better than Huffman coding with low hardware cost (see Section 4).
- We empirically demonstrate that Lane Compression performs reliably with high compression rates on all data sources from seven different neural networks. This includes two different use-cases (inference and re-training[1]), and a wide range of statistical characteristics (see Section 7).

[1]Training on machine learning models that have converged at least once.

- We show that static profiling is sufficient for Lane Compression to perform well, even on data generated dynamically without any further run-time adaptation (see Section 5).
- We outline a hardware implementation of Lane Compression that is lightweight and practical for compressing both on-chip and off-chip communication traffic on resource-constrained hardware. The reconfigurable nature of the hardware ensures that the best compression technique can be selected for the data (see Section 6).

Evaluation of Lane Compression is performed on five machine learning data sources (activations and weights for inference, and activations, weights, and gradients for re-training) from all layers (including convolutional layers, fully connected layers, and subsidiary layers for activation functions, normalisation, and pooling), of seven distinct neural networks (LeNet-5 [35], CifarNet[2], ResNet-18 [20], SqueezeNet [23, 28], MobileNet [21], AlexNet [34], and an LSTM network [7]) for four image or text based datasets (MNIST [35], CIFAR-10 [33], ImageNet [11], and WikiText-2 [38]). Lane Compression targets data obtained during inference and re-training, assuming the statistical properties of data are more stable after convergence, thus more suitable for the proposed method. We demonstrate that Lane Compression provides reliably high performance regardless of data source, layer type, learning model, and input data type, unlike other lightweight compression methods for machine learning. This result implies that the compression rate of Lane Compression is expected to be high without modification or re-implementation of the hardware for new machine learning techniques.

In terms of geometric mean, Lane Compression achieved minimum 93% and maximum 102% (see Section 7 for justification) of the Shannon limit on a per network basis, and minimum 90% and maximum 105% of the Shannon limit on a per data source basis. We implemented Lane Compression in hardware and the total hardware area was 23000$\mu$m$^2$ for the encoder and 18000$\mu$m$^2$ for the decoder in a 40nm process, which is similar in size to a 4KiB SRAM each.

## 2 RELATED WORK

Zero-value compression (ZVC) [46] is a simple but effective lossless compression method for sparse data which stores non-zero values only, with a bit-mask to indicate locations of the non-zeros. ZVC is the only lightweight compression method that does not have an algorithmic parameter and its compression rate is solely dependent on the sparsity of the source data.

Run-length coding on zeros (Z-RLC) [5, 18, 29] and compressed sparse column/row (CSC/CSR) [6, 17, 24, 43, 51] exploit consecutive zeros. Z-RLC is a variation of run-length coding (RLC), counting only runs of zeros instead of all values. The algorithmic parameters for Z-RLC and CSC/CSR decide the window size for counting consecutive zeros. If these parameters are too small then highly sparse data cannot be compressed efficiently, and if these parameters are too large then the overhead of compression becomes significant for less sparse data. Any positive integer parameter can be used for Z-RLC, which makes Z-/RLC the only lightweight compression methods that can compress a sequence of values at once, and parameters for CSC/CSR are capped by sizes of matrices such as feature maps or filters. Unlike Z-RLC, CSC/CSR stores 2-dimensional position information of non-zeros which may hamper the compression rate but can improve efficiency of computation.

Null suppression (NS) [9, 47] is a compression technique that removes leading zero bits from a value. NS is well-suited to machine learning because the majority of machine learning values are small, regardless of the source of data. Sparse Exponential-Golomb (S-EG) [15] eliminates leading zeros of individual values, and Dynamic Precision Reduction (DPRed) [10] eliminates leading zeros common to a block of values. NS methods do not fully rely on sparsity of data unlike prior methods.

[2] A CNN developed internally for CIFAR-10. The model definition can be found at https://github.com/deep-fry/mayo/blob/master/models/cifarnet.yaml

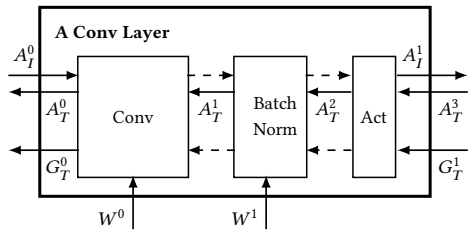



Fig. 2. Dataflow of a convolutional layer showing activations ($A^i$), weights ($W^i$) and gradients ($G^i$) for inference ($_I$) and training ($_T$). Solid arrows represent data that moves between execution units and so could be compressed to save bandwidth.

A few memory compression techniques have been adapted to machine learning such as frequent pattern compression (FPC) [29, 52] and Base-delta-immediate compression (BDI) [45, 52]. Extended bit-plane compression (EBPC) [4] is a variation of FPC designed for machine learning combined with ZVC and Z-RLC. These methods incorporate some data-driven information into compression such as a dictionary of frequent patterns of the source data, or common bases or ranges of deltas in a block of values.

Improved compression rates may be possible with more complex compression algorithms such as Huffman coding [8, 18], LZW [8], and DEFLATE [8, 46], but at a much higher, and often impractical, hardware and run-time cost for embedded systems. Moreover, these methods are designed for general purpose data such as text and numbers which have distinctive and different statistical characteristics to machine learning data.

In this paper, we compare Lane Compression with all compression methods mentioned above except BDI. The method has been demonstrated only on weights to the best of our knowledge, and finding the most frequent bases and ranges of deltas for other data sources is an independent research topic and beyond the scope of the paper.

## 3 BACKGROUND

### 3.1 Compression in Machine Learning

Using machine learning for inference or training involves an enormous number of matrix multiplications and massive memory traffic to supply data for the computation [25]. A convolutional network is composed of multiple convolutional layers with additional pooling layers in between. Figure 2 depicts the baseline dataflow of a convolutional layer for this paper. Left-to-right arrows represent flow of activations ($A_I^0$ and $A_I^1$) during inference and right-to-left arrows represent flow of activations ($A_T^0$, $A_T^1$, $A_T^2$, and $A_T^3$) and gradients ($G_T^0$ and $G_T^1$) during training. Weights for convolution ('Conv') ($W^0$) and the scale and shift parameters for batch normalisation ('Batch Norm') ($W^1$) will be used and updated for inference and training respectively. Activation functions ('Act') usually do not require parameters, or constant parameters if needed. In this paper, we quantise and compress dataflows denoted by solid arrows. The dataflows denoted by dashed arrows indicate that data can be produced and consumed without being stored in memory [14, 39]. For training, any intermediate gradients of activations are stored in memory and gradients of weights are computed internally without being stored. State-of-the-art quantization methods for weights usually quantise $W_0$ only and also exclude the first and last convolutional layers from quantization to minimise any drop in accuracy [12, 26, 54, 58]. Such restrictions are reasonable for resource-heavy devices such as GPU machines but not for resource-constrained devices. Any partially unquantised values require full floating point functional units such as multipliers and adders which is up to an order

of magnitude more expensive in area and energy consumption than fixed-point functional units. In addition, the proportional overhead for batch normalisation is not negligible even on GPU machines [27]. Thus we quantise and compress $W^1$ as well as $W^0$ for all layers.

### 3.2 Lossless Compression

Table 1. Lossless compression methods for machine learning. Target data source column presents on eor more data sources, i.e. activations ($A$), weights ($W$), or gradients ($G$), that have been targeted by each compression method.

| Method | Compression unit | Algorithmic parameter(s) | Coverage | HW feasible | Target data source |
|---|---|---|---|---|---|
| ZVC | Symbol | None | Low | Yes | A, G |
| Z-RLC | Sequence | Max sequence | Low | Yes | A |
| CSC/CSR | Symbol (block) | Size of matrix | Low | Yes | A, W, G |
| NS | Symbol | Size of dense bits | Med | Yes | A |
| | Symbol (block) | Size of block | Med | Yes | A, W, G |
| FPC | Symbol (block) | Patterns | Med | Yes | A, G |
| BDI | Symbol (block) | Base(s) and delta | Med | Yes | W |
| Huffman | Symbol | Huffman code table | High | No | W |
| LZW | Symbol, Sequence | Size of code table, code table | High | No | W |
| DEFLATE | Symbol, Sequence | Size of window, code table | High | No | A, W |
| Lane (ours) | Symbol, Sequence | Bit-split, methods | High | Yes | A, W, G |

Data compression exploits statistical characteristics of source data to reduce data redundancy. One of the commonly used statistical characteristics is the probability distribution of symbols. Symbols refer to the base unit of data such as characters in text or quantised integer values in machine learning. Compressing source data with incompatible statistical characteristics to the applied compression method may cause compression to fail and data after compression can even be larger than the source.

This subsection presents the three factors that we have identified that define the characteristics of lossless compression. Table 1 classifies the lossless compression methods discussed in Section 2 in terms of these three factors.

*3.2.1 Compression unit:* the unit of source data to be passed into the compression function. Commonly used compression units are individual symbols or sequences of symbols. The probability distribution of the compression unit determines the compression rate. Huffman coding is a well-known symbol-based compression method, and arithmetic coding [44] and Lempel-Ziv-Welch coding (LZW) [53] are popular sequence-based compression methods. Symbol-based compression methods assume symbols are independent and identically distributed (iid), and sequence-based compression methods consider mutual dependencies between symbols in addition. Not assuming iid may result in better compression, however the probability distribution of sequences is much more complex than that of symbols, typically causing high compression overhead.

Machine learning data is multi-dimensional, so different ways to serialise the data before compression may affect the amount of mutual dependence between symbols. In this paper, we serialised the data by the commonly-used NCHW (batch-channel-height-width) order [42].

*3.2.2 Coverage:* the possible probability distributions that can be effectively handled by a compression method. Restricting coverage can improve the compression rate in particular cases or reduce the implementation complexity, at the cost of generality. Both Huffman coding and ZVC are symbol-based compression methods, but ZVC limits its coverage to highly sparse data distributions whereas Huffman coding covers all distributions. This coverage limitation allows ZVC to achieve high compression rates without any additional overheads such as Huffman tables, but only if the redundancy of the source data is mainly in zeros. ZVC commonly fails to compress otherwise.

*3.2.3 Hardware Feasibility:* hardware implementation cost is highly sensitive to the amount of state maintained within compression algorithms. Lightweight compression methods, marked as hardware feasible in Table 1, require only a small number of parameters whereas the complex compression methods such as Huffman coding, LZW, and DEFLATE, require a large table of parameters to map each symbol or sequence of symbols to its codeword. One of the distinctive merits of complex compression methods against lightweight compression methods is their high adaptivity to dynamic input data, and this makes hardware implementation even more complicated. Huffman coding is not naturally adaptive, but can gain adaptivity by rebuilding the Huffman table on the fly or by speculating statistics of the source. Either approach significantly increases the implementation complexity and may or may not benefit compression rate [32]. LZW is a universal compression method and does not require a priori knowledge of the statistics of the source, but its dictionary needs to be rebuilt on every encoding/decoding operation and the size of the dictionary can be even larger than a similar Huffman table. DEFLATE uses a combination of Huffman coding and LZSS which is a version of LZ compression. The hardware implementation complexity of arithmetic coding is also high due to high precision computations, although this can be partially mitigated using approximate computing or lookup tables [48].

In Section 5.2, we have demonstrated that statistical fluctuation in dynamic machine learning data for inference and re-training is rather modest, which might not compensate for the high cost of adaptivity in previously mentioned complex compression methods.

### 3.3 Lossy Transformation

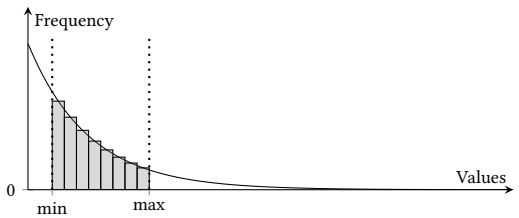



Fig. 3. Illustration of lossy transformation for machine learning [26]: pruning (min), clipping (max), and quantization (bars between min and max).

A lossy transformation applied to a machine learning model minimises the amount of information without harming the overall model accuracy. Popular lossy transformation techniques include pruning, clipping and quantization as shown in Figure 3.

Pruning is a technique to reduce the amount of information by selectively converting non-zero values to zero. Values smaller than a threshold (denoted as 'min' in Figure 3) are converted to zero,

increasing sparsity of the data. Clipping reduces the amount of information by converting values larger than a threshold (denoted as 'max' in Figure 3) to that threshold. Quantization is a way of binning data into $2^b$ bins (denoted as bars in Figure 3) where $b$ is the bitwidth of the target data representation after quantization. Quantization effectively reduces the amount of information by reducing the resolution of individual symbols. Quantization conveys a similar effect to pruning for values smaller than the minimum resolution. Clipping is frequently accompanied by quantization to achieve higher resolution of data for a given bitwidth.

We use machine learning data that are linearly quantized without clipping in the rest of this paper, except Section 7.1. In Section 7.1, we discuss the effectiveness of Lane Compression with different levels of lossy transformation, to show our approach is not limited to a specific lossy transformation.

### 3.4 Compression Rate

The overall compression rate comprises compression rates from lossy transformations ($Q$), and lossless compression ($C$),

$$\text{overall compression rate} = Q \times C.$$

We use 32-bit floating point as the full precision representation, thus $Q = \frac{32}{B}$ where $B$ is the bitwidth after quantization. Note that sign bits are discarded if values are analytically non-negative (e.g., activations after ReLU activation functions). For such layers, $Q = \frac{32}{B-1}$.

Since Lane Compression is a lossless compression technique, all the numbers provided in this paper correspond to $C$ in the equation above, not the overall compression rate if not specified. Compression rate $C$ can be combined with any lossy transformation and/or quantization method to amplify the overall compression rate.

## 4 LANE COMPRESSION

The intuition behind Lane Compression is that different portions of values seen in deep machine learning have different characteristics. The most significant bits tend to be very sparse, with sparsity steadily reducing as the bits become less significant. We therefore split values into multiple *lanes* and apply a tailored compression method to each lane.

Table 2 shows statistical characteristics of machine learning data from a selected set of networks. Entropy density is the Shannon marginal entropy [49] over the bitwidth of the data type and represents how many bits in the data contain information. The main redundancy in machine learning data is sparsity in both values and bits [41]. The sparsity in values is the proportion of zero-valued symbols in a dataset, and the sparsity in bits is the proportion of zero-bits in a signed magnitude binary representation. In addition to many machine learning data sources having high value-level sparsity, even dense data has high bit-level sparsity as shown in Table 2. This is because most non-zero machine learning data values are very small regardless of the source. The intuition of our approach is that the hidden bit-level sparsity can be exposed by splitting values into bit segments, or lanes.

For the inference results shown, we train models using full-precision floating point values, then quantise to the minimum bit widths while reducing accuracy by no more than 0.5%. For re-training, we train models from scratch using quantised values with the minimum bit widths that converge to the same or better accuracy within the bound as the baseline full-precision floating point models. Since this quantization level is sufficient for complete training, we believe it to also be sufficient for fine-tuning pre-trained models.

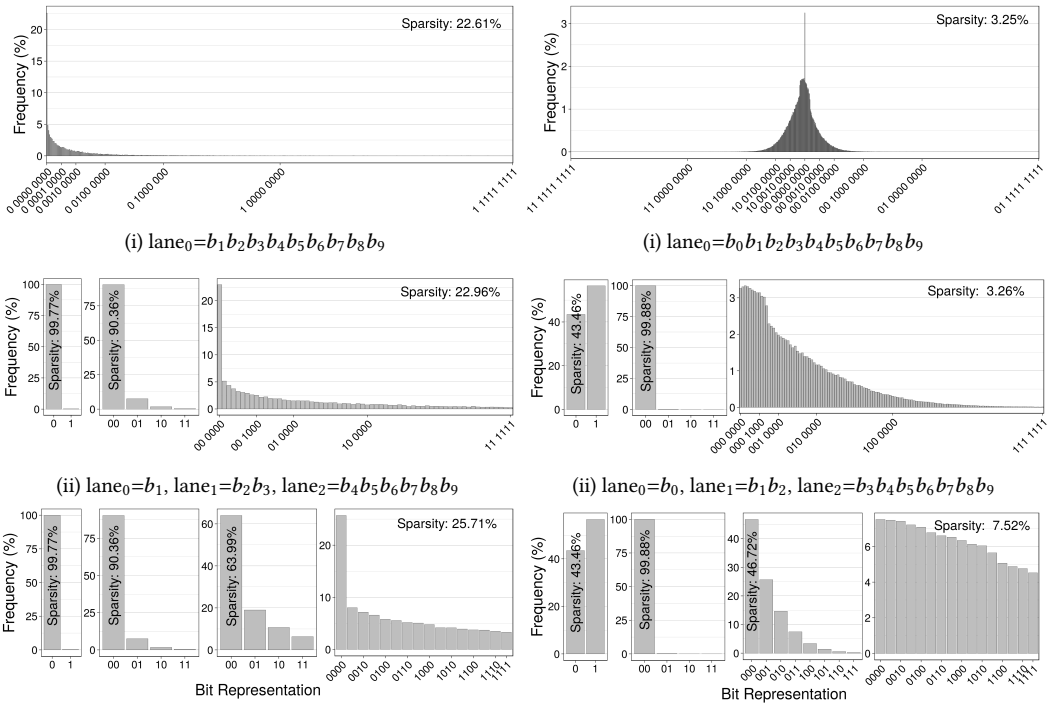
Figure 4 visualises how data characteristics change by splitting bits into lanes. Value sparsity of the original activations is 22.61% (Figure 4(a)(i)). As each value is split into lanes, the sparsity

Table 2. Machine learning data characteristics from inference (*Inf.*) and re-training (*Re-tr.*). *Data type* refers data representation after quantization in fixed-point (FxP). Top-1 model accuracy is provided with differences to the full precision model accuracy in parentheses.

| Network | Inf./ Re-tr. | Data source | Data type | Value sparsity | Bit sparsity | Entropy density | Top-1 model acc. |
|---|---|---|---|---|---|---|---|
| LeNet-5 | Inf. | Acts | 10 FxP | 1.51% | 65.20% | 63.99% | 99.20% (-0.04%) |
| | | Weights | | 0.82% | 61.37% | 82.54% | |
| | Re-tr. | Acts | 16 FxP | 1.03% | 60.51% | 73.88% | 99.14% (-0.10%) |
| | | Weights | | 15.81% | 67.70% | 70.37% | |
| | | Grads | | 4.96% | 76.37% | 56.62% | |
| CifarNet | Inf. | Acts | 10 FxP | 65.54% | 90.00% | 29.52% | 93.23% (+0.02%) |
| | | Weights | | 2.35% | 66.28% | 75.65% | |
| | Re-tr. | Acts | 16 FxP | 24.02% | 71.77% | 60.91% | 92.98% (-0.23%) |
| | | Weights | | 40.42% | 76.49% | 55.53% | |
| | | Grads | | 10.26% | 79.07% | 52.50% | |
| ResNet-18 | Inf. | Acts | 12 FxP | 36.33% | 77.64% | 50.42% | 69.25% (+0.01%) |
| | | Weights | | 0.94% | 66.95% | 72.12% | |
| | Re-tr. | Acts | 24 FxP | 18.94% | 65.88% | 69.72% | 69.53% (+0.29%) |
| | | Weights | | 0.09% | 61.50% | 78.07% | |
| | | Grads | | 29.77% | 75.19% | 54.58% | |
| MobileNet | Inf. | Acts | 14 FxP | 54.40% | 83.24% | 38.93% | 68.12% (-0.03%) |
| | | Weights | | 33.84% | 76.78% | 56.75% | |
| | Re-tr. | Acts | 24 FxP | 41.05% | 75.32% | 52.95% | 68.97% (+0.82%) |
| | | Weights | | 27.32% | 71.90% | 56.87% | |
| | | Grads | | 34.60% | 78.37% | 50.60% | |
| LSTM | Inf. | Acts | 12 FxP | 1.24% | 69.85% | 60.41% | 23.09% (-0.02%) |
| | | Weights | | 5.14% | 77.28% | 54.96% | |
| | Re-tr. | Acts | 20 FxP | 1.21% | 64.12% | 74.73% | 23.08% (-0.03%) |
| | | Weights | | 1.94% | 68.50% | 70.72% | |
| | | Grads | | 18.35% | 92.62% | 20.42% | |

of the most significant lane increases drastically and the entropy of the source is shifted to the least significant lane. A similar tendency is observed for the weights, even though their value sparsity (3.25%) is significantly lower than the activations. Since information content and sparsity are not distributed evenly across the bits, we can break values into lanes, each with more distinctive characteristics. We can then take advantage of this to apply more-specialised compression methods to each lane. The least significant lane has slightly less entropy than the source in far fewer bits which means the least significant lane has much higher density of information ($= \frac{\text{entropy}}{\text{bit width}}$) than the source. Data with higher density of information is harder to compress but the overall compression rate is improved because of the drastic increase of sparsity in the other lanes.

After splitting into lanes, each lane is compressed separately as illustrated by the example in Figure 5. Every lane has distinct statistical characteristics from other lanes. Thus the best compression method and parameter are chosen for each lane to harvest different forms of redundancy. In this example, the first lane ($\text{lane}_0$) is the most sparse lane where Z-RLC with a wide 12-bit counter




(i) $\text{lane}_0{=}b_1b_2b_3b_4b_5b_6b_7b_8b_9$ (i) $\text{lane}_0{=}b_0b_1b_2b_3b_4b_5b_6b_7b_8b_9$

(ii) $\text{lane}_0{=}b_1$, $\text{lane}_1{=}b_2b_3$, $\text{lane}_2{=}b_4b_5b_6b_7b_8b_9$ (ii) $\text{lane}_0{=}b_0$, $\text{lane}_1{=}b_1b_2$, $\text{lane}_2{=}b_3b_4b_5b_6b_7b_8b_9$

(iii) $\text{lane}_0{=}b_1$, $\text{lane}_1{=}b_2b_3$, $\text{lane}_2{=}b_4b_5$, $\text{lane}_3{=}b_6b_7b_8b_9$ (iii) $\text{lane}_0{=}b_0$, $\text{lane}_1{=}b_1b_2$, $\text{lane}_2{=}b_3b_4b_5$, $\text{lane}_3{=}b_6b_7b_8b_9$

(a) Activation (b) Weight

Fig. 4. Changes in data characteristics by splitting bits of (a) activations and (b) weights from a layer of CifarNet for inference. Each graph shows the distribution of non-zero values, and the value sparsity is given in the top-right corner.

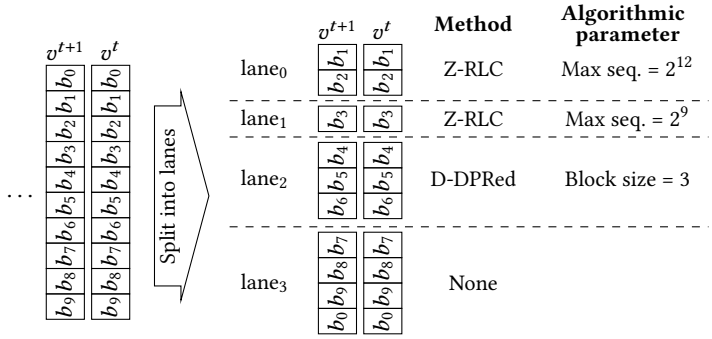



Fig. 5. Lane Compression with an example configuration. Values ($v^i$) in the input stream comprised of 10 bits ($b_i$) each are split into lanes ($\text{lane}_i$) of smaller bitwidth before being compressed. The choice of splitting and method is determined by ahead of run-time profiling.

Table 3. Compression techniques considered for compressing individual lanes

| Method | Algorithmic parameter | Range ($p \in \mathbb{N}$) |
|---|---|---|
| None | None | N/A |
| ZVC | None | N/A |
| RLC | Max seq. ($2^p$) | $1 \leq p \leq 32$ |
| Z-RLC | Max seq. ($2^p$) | $1 \leq p \leq 32$ |
| D-DPRed | Block size ($p$) | $1 \leq p \leq 8$ |
| S-DPRed | Block size ($p$) | $1 \leq p \leq 8$ |

is most effective. The second lane ($\text{lane}_1$) is also highly sparse but not as much as the first lane. Thus Z-RLC with smaller 9-bit counter is most suitable for the second lane. Later lanes eventually become too dense to exploit simple sequence based compression methods. The third lane ($\text{lane}_2$) still has some symbol level redundancy, thus D-DPRed, a variation of DPRed for dense data, is used on every block of three symbols. The last lane ($\text{lane}_3$) does not have sufficient data redundancy to compensate for the overhead of any compression method, and so we do not compress the last lane. Note that sign bits, $b_0$, are highly random in general and have no correlation to neighbouring bits, thus sign bits have been shifted to the least significant bit position. Sign bits are not used if values are analytically always positive, such as activations after ReLU, as shown in Figure 4(a).

The compression methods used to compress individual lanes and their parameters are listed in Table 3. Sparse-DPRed (S-DPRed) and Dense-DPRed (D-DPRed) are variations of DPRed that we propose, and they are specialised for dense and sparse data respectively. For S-DPRed we add one more bit to the original DPRed to indicate whether a block is full of zeros or not, and for D-DPRed the zero-bit mask is dropped. The original DPRed works well as a compromise on common datasets which are composed of sparse or dense values, but either D-DPRed or S-DPRed is better when a dataset has consistent characteristics. Therefore these DPRed variations outperformed the original DPRed when used in lanes because of the ahead of run-time profiling (see Figure 10(b) in Section 6).

Lane Compression has flexibility to choose the best bitwidth to compress, and also to choose the best sequence-based or symbol-based method for each lane depending on the type of the data redundancy. Such flexibility provides high coverage, which is specifically important for the machine learning domain where new techniques are introduced regularly. A compression method with high coverage means high compression rates are expected without modification or re-implementation for the new techniques.

## 5 PROFILING AND EXPLORATION

The best split of lanes and the best methods and parameters per lane are decided by profiling data statically for both static and dynamic data. The profiling step explores all possible configurations exhaustively. This section explains the profiling step in detail and also feasibility and quality of profiling.

### 5.1 Search Space

Finding the best configuration for Lane Compression is a combinatorial problem. The size of the search space $S$ is,

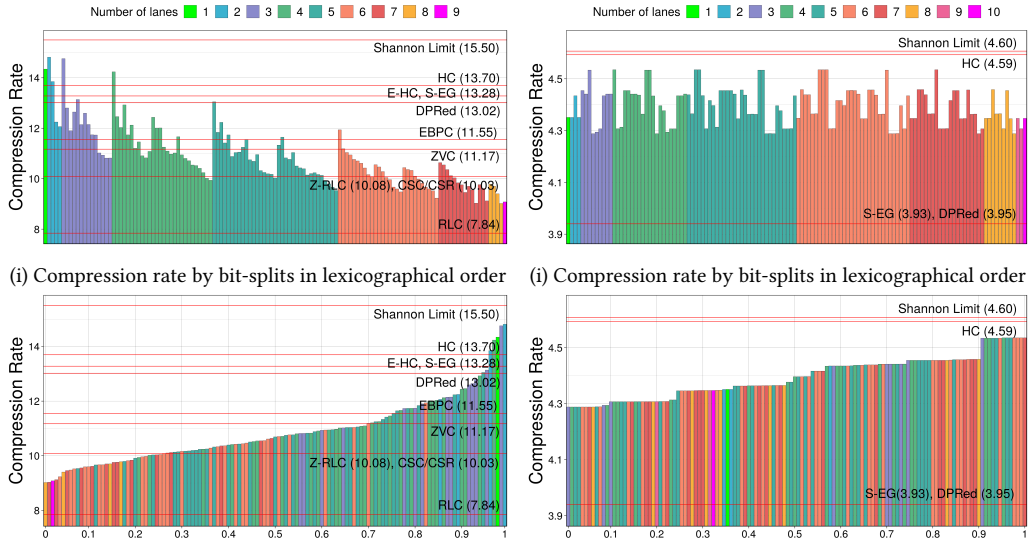
$$|S| = \sum_{k=1}^{n} \left( \binom{n-1}{k-1} \left( \sum_{m \in M} |P_m| \right)^k \right)$$

where $n$ is the bit width of data, $k$ is the number of lanes, $M$ is the set of available methods, and $P_m$ is the set of available parameters for a method $m$ for $m \in M$. Therefore $\sum_{m \in M} |P_m| = 1 + 1 + 32 + 32 + 8 + 8 = 82$ for the configuration used in the paper (see Table 3). For instance, $|S| = 5.011 \times 10^{30}$ for 16 fixed-point (FxP) data. To reduce the search space, we assume that individual lanes are independent from each other. This assumption allows us to find the best method and parameter for every possible width and position of lanes first and then combine them to estimate the performance of each split of lanes. With this assumption, the size of the alternative search space $S'$ becomes,

$$|S'| = \sum_{l=1}^{n} l \sum_{m \in M} |P_m| + \sum_{k=1}^{n} \binom{n-1}{k-1}$$

where $\sum_{l=1}^{n} l$ is the number of every possible width and position of individual lanes. This assumption reduces the search space drastically, making exhaustive search feasible. For instance, $|S'| = 43920$ for 16 FxP data.

This assumption is correct theoretically but not in practice, because our hardware implementation of Lane Compression exploits synchronous behaviour of individual lanes to cope with infinite bit streams using a very small buffer (see Section 6.1). We argue that the huge reduction in search space under this assumption outweighs the minimal loss in compression rate.




(i) Compression rate by bit-splits in lexicographical order

(ii) Bit-splits ordered in compression rate

(a) Activation

(i) Compression rate by bit-splits in lexicographical order

(ii) Bit-splits ordered in compression rate

(b) Weight

Fig. 6. Overall compression rate ($Q \times C$) of different bit splits vs. 32-bit in (i) lexicographical order and in (ii) order of compression rate for (a) activations and (b) weights from a layer of CifarNet for inference. Red horizontal lines are provided to compare with other compression methods and the Shannon limit.

Figure 6 shows the compression rate of all possible splits of the activations (Figure 6(a)) and weights (Figure 6(b)) used for Figure 4. Figure 6(a)(i) and Figure 6(b)(i) show compression rates for different splits in lexicographical order (i.e., splits on more significant bits precede the splits on less significant bits when the number of lanes is same.), and Figure 6(a)(ii) and Figure 6(b)(ii) show the same results in order of compression rate to present likelihood to achieve a better compression rate

than other compression methods. The left-most bar in Figure 6(a)(i) shows the compression rate without splitting and the right-most bar shows compression after splitting every bit into its own lane. It is worth noting that splitting bits into too many or too few lanes harms compression rate. Insufficient bit-level redundancy may be exposed if the number of lanes is too small, and if splitting bits does not reveal further data redundancy then it only adds compression overhead without gain.

We use static Huffman coding for comparison, where the Huffman table contains only codewords of the observed values. The Huffman table has to be updated if an unknown value is encountered which not only increases the size of the dictionary but may also increase the length of codewords of other values, lowering the overall compression rate. We did not include this overhead for adaptivity in the comparison. Thus the compression rate of static Huffman coding for dynamic data, e.g. 13.70× for activations, is only a maximum possible compression rate. The result of Exhaustive Huffman coding (E-HC), which is 13.28× and covers all possible values for a given data representation, is provided in addition to suggest a realistic compression rate including the overhead for adaptivity.

For activations, every possible split performed better than RLC, 73.8% of splits performed better than Z-RLC and CSC/CSR, 28.9% of splits performed better than ZVC, 5.1% of splits performed better than DPRed, and 3.1% of splits performed better than E-HC and Huffman coding. For weights, every possible split performed better than all lightweight compression methods. CSC/CSR, ZVC, RLC, and Z-RLC failed to compress weights resulting in larger compressed data than the source. Thus the results are omitted from the graphs. Static Huffman coding performed best on weights, achieving 4.59×, which is close to the Shannon limit, but at the cost of around 1KB memory space for the Huffman table only for this layer and the need to access this memory repeatedly to encode and decode data. Lane compression (4.54×) achieved 98.6% of the limit without a dictionary.

### 5.2 Profiling Dynamicity

Profiling static data ahead of run-time is relatively straight-forward, but not all machine learning data is static. In this section, we show that the statistical characteristics of machine learning data that Lane Compression exploits are strong enough even in dynamic data, such as activations and gradients, to be profiled ahead of run-time.

First we use the Z-test and the central limit theorem[3], on 100 random samples from different batches to estimate the performance of Lane Compression on the population of dynamic data. One sample corresponds to all activations or gradients generated by one input to the network (for example, activations from all layers from one input image into CifarNet).

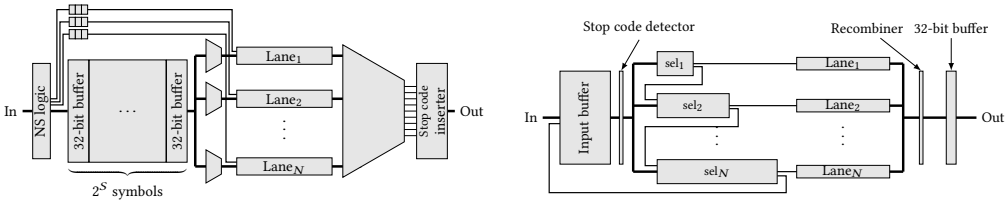
The 95% confidence interval (CI) of the compression rate of Lane Compression on dynamic data showed less than 1.75% difference for on activations and less than 3.6% difference on weights over mean of the 100 samples. This indicates that the compression rate does not change much even with different inputs. This also implies that the high performance is expected not to be degraded by unseen input data such as activations or gradients of new images. This also implies the statistical characteristics exploited by lossless compression are determined by the network and not by its input. Gradients show relatively wider intervals because gradients have higher sequence based data redundancy than other data sources, leading to high but also slightly more fluctuating compression rate.

Table 4 shows the results of another experiment, the quality of configurations for Lane Compression after different amounts of profiling compared to the best configuration of the evaluation set. We randomly partition the dataset into a profiling set and evaluation set. In this experiment, we investigated the minimum size of the profiling set needed to search for the best configuration.

[3]Standard deviation of random samples greater than 30 in size is considered as a valid standard deviation of the population based on the central limit theorem.

Table 4. Quality of profiling dynamicity by different sizes of profiling set

| | 1 | 2 | 4 | 8 | 16 | 32 |
|---|---|---|---|---|---|---|
| Inf. Acts | 97.68% | 99.66% | 99.87% | 99.95% | 99.83% | 99.99% |
| Re-tr. Acts | 99.63% | 99.75% | 99.95% | 99.95% | 99.97% | 99.99% |
| Re-tr. Grads | 94.24% | 97.73% | 99.40% | 99.71% | 99.84% | 99.90% |




(a) Encoder (b) Decoder

Fig. 7. Lane Compression block diagram. Blocks sized proportional to approximate hardware area cost.

Notably, a configuration selected by profiling a single sample could achieve a minimum of 94.24% of the compression rate of the best configuration. This indicates that statistical characteristics of machine learning data are consistent enough to be detected by only a single sample. For this paper, we profiled once per dynamic data source using four random samples for each.

## 6 HARDWARE IMPLEMENTATION

Lane Compression was designed with a simple hardware implementation in mind. In this section we describe a reconfigurable implementation of Lane Compression, as we aim to support a wide range of machine learning networks with a single hardware design. This design would be suitable for use on an ASIC, either situated at the memory controller or instantiated repeatedly within an accelerator to compress and decompress on-chip traffic and memory.

In order to match realistic design constraints, we present a version of a Lane Compression encoder which produces a single fixed width output bitstream, and symmetrically a decoder which receives a single fixed width input stream. The design would be simpler if each encoder lane produced a separate stream that was never recombined, but the rest of the system would then have to be modified to handle these multiple streams. This may be appropriate for some use cases, but we choose to focus on the more generic and challenging case. Producing a single fixed width input stream is the source of most of the complexity in our design, consuming over half the area.

Figure 7 shows a block diagram for the encoder and decoder. The blocks in the diagram are sized proportionally to their approximate hardware cost in area. On the left side of the encoder is the NS logic. This block detects the number of leading zeroes in groups of lane values to support S/D-DPRed. The leading zero counts are transmitted to all S/D-DPRed lanes. The input symbol then queues in a FIFO of size $2^S$. The lanes are simple, reconfigurable implementations of the compression methods, with the constraint that every input symbol generates exactly one (possibly null) variable length output symbol. The output symbols of the lanes must then be interleaved by muxing logic to form combined output symbols for the whole encoder. This is done by shifting each output value to the appropriate offset and performing a bitwise or operation. Finally, the combined symbols must be checked for an escape sequence known as a stop code (see Section 6.1). If the symbol matches a stop code, an extra bit must be inserted to prevent ambiguity in the decoder.

The decoder is similar to the encoder but in reverse. Input compressed symbols are buffered in the input buffer. This must be large enough for the worst case compressed symbol. A stop code detector checks for stop code sequences and handles them appropriately. It is important for the decoder's throughput to quickly compute the size of each variable-length compressed symbol because the next symbol decode cannot start until the width of the previous symbol is known. Therefore, all possible symbol widths are precomputed on the previous cycle. This precomputation explores more possibilities as the number of lanes increases, which is why the selection logic gets bigger in the figure for each lane. Once the symbol is available, the symbol is inspected by each lane sequentially in order to deduce the true symbol width (see Section 6.2). Once all lanes have inspected the data, the symbol width is known and the appropriate number of bits removed from the buffer. The symbol is then split up, reversing the combination logic in the encoder. This is achieved by shifting and masking the appropriate number of bits from the symbol. The symbol for each lane is decompressed and then passed to the recombining logic which shifts the outputs to the correct position to form the full output symbol. This final recombination logic is simple because the shift and mask values are static for a given configuration.

Note that our hardware design trades increased encoder complexity for a simple decoder. A lightweight decoder is useful when multiple decoders per encoder are used, for instance, when a value needs to be read multiple times but the locality of the value is not high enough to compensate the cost of local storage.

### 6.1 Bit Packing

This section describes how our hardware implementation creates a fixed width stream for the compressed data, despite the variable width nature of the individual lane methods.

A simple naive scheme is to concatenate the results of each encoder lane to form a single variable width output symbol in the encoder, and symmetrically to split the concatenated values in the decoder. Unfortunately this naive scheme only works for symbol-based lane methods. Z-RLC and RLC are sequence-based methods and thus not every input symbol generates an output symbol. A fundamental aspect of this problem is that in order to generate any output symbol a (Z-)RLC encoder needs to receive *all* the input data for that run. On the other side, the decoder needs to receive the output symbol before it can decode *any* information about that run. Thus the encoder produces information late, but the decoder needs it early. This is a problem for Lane Compression, as we need to synchronise the output data from each lane.

Our solution is to modify the (Z-)RLC methods to introduce the concept of 'long/short runs'. We introduce a parameter $S$ and define a run of length $2^S$ or more as 'long' and otherwise 'short'. Short runs are encoded in a standard way, a value followed by the run length in $S$ bits, and long runs are encoded as the value followed by $2^S - 1$. The encoder must then emit a stop code to signal when the long run ends. This ensures that the encoder only needs to see at most the next $2^S - 1$ input symbols before outputting an output symbol, thus only a buffer of size $2^S$ is needed in the encoder as shown in Figure 7.

The stop code for long runs should be recognisable to the decoder and be unique for correctness. We introduce a parameter $C$ for the width of the stop code. The value of the stop code is chosen as $100\ldots00$; a 1 bit followed by $C - 1$ 0 bits. This must then be followed by an additional 0 bit if the stop code is real or a 1 bit if it is spurious. Spurious stop codes are cases where the output symbol happens to match the stop code value. If there is a spurious stop code, the decoder simply deletes the extra 1 bit and carries on processing normally. We have carefully chosen the value of the stop code for simplicity of the hardware implementation and to reduce the number of spurious occurrences. After the stop code, it may be necessary to indicate which lane stopped if there is

more than one (Z-)RLC lane. Thus, the index of the lane should be emitted, but the width of this index value can be less than $\log_2(\text{lanes})$ if there are few (Z-)RLC lanes.

One consequence of our modification of (Z-)RLC is that there must be at least one lane which is symbol based. This is because the stop code encoding implicitly uses the other lane's data to synchronise when to stop the run.

```
while True:
  b = []
  for l in Lanes:
    if l.need_stop_code:
      output('10..0', '0', l.index)
    b += l.output_symbol
  output(b[0:C])
  if b[0:C] == '10..0':
    output('1')
  output(b[C:])
```

Fig. 8. Pseudo-code for Bit Packing algorithm

Figure 8 shows pseudo-code for our bit packing hardware. Not shown is the fact that spurious stop codes may span multiple iterations of the 'while' loop, so it is necessary to detect this case as well. Stop codes can only be generated at the start of a concatenated output symbol and so consequently the decoder only needs to check for stop codes at the start of a concatenated input symbol.

| Processing Order | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| Input | 00000 | 00001 | 00010 | 00011 | 00000 | 00100 | 01000 |
| Lane 0 in | 00 | 01 | 10 | 11 | 00 | 00 | 00 |
| Lane 1 in | 000 | 000 | 000 | 000 | 000 | 001 | 010 |
| Stop code | | | | | | **10** | |
| Lane 0 out | 0 | 101 | 110 | 111 | 0 | 0 | 0 |
| Lane 1 out | 00011 | | | | | 001 | 010 |
| Concatenation | 000011 | **10**1 | 110 | 111 | 0 | **10**0001 | 0010 |
| Output | 000011 | **101**1 | 110 | 111 | 0 | **100**0001 | 0010 |

Stop code: **10**
Methods: Lane 0: ZVC
Lane 1: Z-RLC, Max seq. = $2^2$

Fig. 9. Example output of Lane Compression. The figure shows inputs and corresponding outputs of the encoder in processing order starting from Column '0'. Within each processing step, bit columns are aligned vertically to denote how the input is split (Rows 'Lane 0 in' and 'Lane 1 in'), how the concatenation is formed (Rows 'Stop code', 'Lane 0 out', and 'Lane 1 out'), and how the real stop codes are distinguished with the spurious ones (Rows 'Concatenation' and 'Output').

Figure 9 shows an example of Lane Compression. In this example a 5 bit input is being compressed with 2 lanes with parameters $S = 2$ and $C = 2$. Lane 0 is using the ZVC method to compress the two least significant bits. Lane 1 is using the Z-RLC method to compress the three most significant bits. For each input, lane 0 (the ZVC lane) just outputs `0` if the input is `00` or the value concatenated with 1. This matches the normal behaviour of a ZVC encoder. For lane 1, the first five symbols are `000`, so this is the start of a run of length 5. This is larger than $2^S - 1 = 3$ and so a long run is

started. This is encoded with a run length of 11 and will eventually be stopped with a stop code. After the run, the remaining inputs to lane 1 are non-zero, so these are encoded directly, matching the normal behaviour of Z-RLC. For each input, all of these encoded values are concatenated ready to be output. At the second input symbol, the concatenated output matches the stop code value `10` (as shown in bold) spuriously and so an extra 1 bit is inserted by the stop code detector. This allows the decoder to know this is not a real stop code. At the sixth input symbol, the run in lane 1 stops, so a stop code is emitted before the output symbol, again shown in bold.

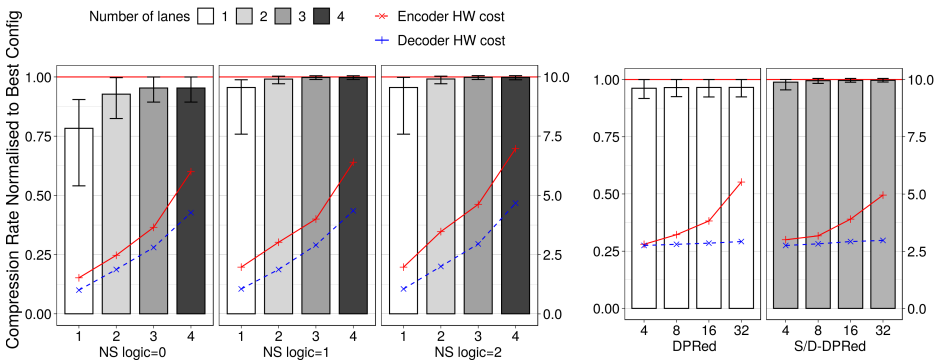
### 6.2 Speculation in Decoder

When using a combined bitstream, the lane decoder hardware has a fundamental speed bottleneck. The compressed bitstream has no annotations or indication as to the boundaries between variable-length symbols, so the decoder cannot begin decoding the next symbol until the end position of the previous symbol is computed. To do this, the decoder must inspect the symbol at the head of the stream and determine the width of each lane's contribution to that symbol. If solved naively, this problem can prove to be a significant bottleneck on the decoder speed. Fortunately, in the vast majority of cases the width of the next symbol can only be one of two possibilities for each lane. This allows speculation of the next symbol width and some information to be precomputed.

For example, in Figure 9, lane 0 is a 2 bit ZVC lane and so each symbol this lane generates is either a 1 bit zero value, or a 3 bit non-zero value. Lane 1 is a 3 bit Z-RLC lane and so on the first input symbol the width will either be a 3 bit non-zero value or a 5 bit run start. Once a run start is seen, the next symbol width will be 0 bits until the run end. Simple rules such as this always give at most two possible widths of all lane symbols before the symbol is read. The only exception is the first symbol of a DPRed block, which cannot be predicted.

Predicting the width of compressed lane symbols allows the width of the whole symbol to be predicted as one of $2^{\text{lane count}}$ possibilities. These width values are precomputed and stored. In all methods, it is sufficient to test if only a subset of bits in the input symbol are all zero in order to select the width. For example, in ZVC, the least significant bit is tested against zero. In the rare case of a symbol width which is not predictable, the decoder stalls for a cycle to wait for the final symbol width to be computed. This system of prediction shortens the critical path of the decoder, allowing it to operate at speed.

### 6.3 Hardware Cost




(a) Number of lanes and NS logic blocks (b) Size of buffers

Fig. 10. Changes in compression rate and hardware cost by optimising reconfigurability.

As reconfigurability adds to hardware cost, those lane widths and compression techniques supported in hardware should be chosen carefully. Experimentation showed that many options could be removed with a minimal loss in compression rate.

We optimised reconfigurability from coarse-grained options (e.g. number of lanes) to fine-grained options (e.g. size of buffer). This is because coarse-grained options are more influential on hardware cost and also on compression rate as shown in Figure 10. Each bar shows the geometric mean compression rate normalised to unconstrained hardware, with an error bar to show minimum and maximum across all data from our benchmark machine learning networks. Relative hardware costs for the encoder and decoder are provided for each design point in addition.

Increasing the number of lanes improved the compression rate and decreased the degree of fluctuation. The area overhead due to a single NS logic block was insignificant compared to the gain, but having two NS logic blocks was not worth the increased hardware cost. Thus we have chosen 3 lanes with 1 NS logic block. Figure 10(b) shows the effect of reducing sizes of buffers (parameter $S$) from $2^5$ to $2^2$. As the buffer size reduces, the hardware cost drops visibly but the compression rate difference is relatively small between $2^4$ and $2^3$ considering both encoder and decoder. Thus we have chosen 3 for $S$. Dynamic data can be more sensitive to buffer size than static data, resulting in a slightly lower compression rate than that of static data. But the gap was negligible i.e. around 0.1% of the overall compression rate for the buffers larger than $2^2$ symbols. Figure 10(b) shows our S/D-DPRed performs better than DPRed as a part of Lane Compression by the aid of profiling. We evaluated the effect of different bit widths of stop codes (parameter $C$) up to 8 bits in hardware cost and compression rate and found the difference is negligible. Thus we have chosen 8 for $C$.

To evaluate the hardware cost we designed an encoder and decoder pair and placed and routed them for the a commercial low leakage power 40nm ASIC process. We targeted a clock frequency of 200 MHz with moderate circuit-level optimisations. Users may apply further circuit-level optimisations, change target hardware area, or use different transistor technologies to match the target throughput of their use cases. The maximum input symbol width and output symbol width were both fixed to 32 bits. Our hardware implementation requires 57 parameters regardless of input data representation and run-time adaptation can be performed by changing these parameters at a cost of 14 cycles if necessary. All lanes support the 'none' and ZVC methods. In addition, Lane 0 supports sparse and dense DPRed, and all other lanes support RLC and Z-RLC. Each lane can also be disabled and the bits each lane compresses can be reconfigured to be any contiguous range of bits from the input. All the reconfiguration state of the hardware is kept in registers. The total area for the encoder was 23000$\mu$m$^2$ and the total area for the decoder was 18000$\mu$m$^2$.

Our hardware implementation costs 8–20 pJ/symbol for encoding and 10–15 pJ/symbol for decoding on a commercial low leakage power 40nm ASIC process. The energy costs are affected by compression rates, showing higher energy efficiency when compression rates are high, because the hardware will have less transitions if the compression rate is good. The current implementation supports up to 32-bit symbols and the energy cost does not vary with the effective bitwidth of inputs. Thus the energy consumption can be drastically reduced by constraining the maximum bitwidth, for instance to 16-bit or 8-bit symbols.

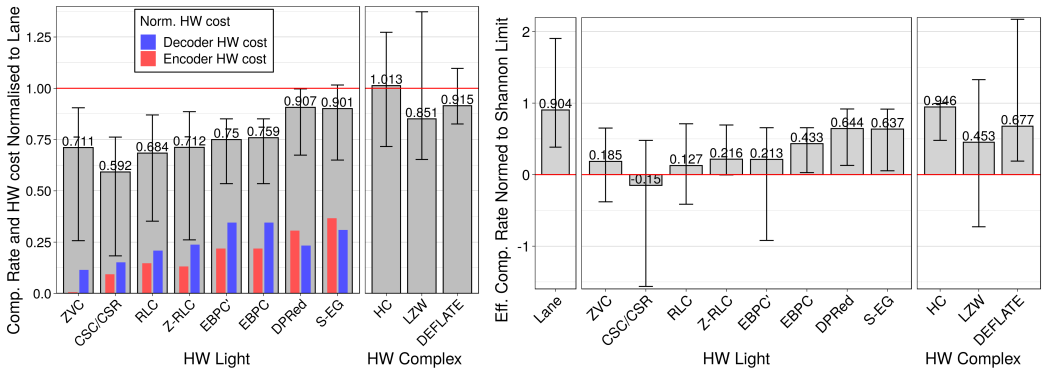
The energy saving from compressing memory traffic can be described as,

$$\frac{\text{energy cost before compression}}{\text{energy cost after compression}} = \frac{n \times b \times E_{mem}}{n \times \frac{b}{c} \times E_{mem} + n \times E_{comp}} = \frac{b \times E_{mem}}{\frac{b}{c} \times E_{mem} + E_{comp}} \tag{1}$$

$$= \frac{c}{1 + \frac{c}{b}\frac{E_{comp}}{E_{mem}}} \tag{2}$$

where $n$ is the number of symbols to be transmitted to complete a workload, $b$ is the bitwidth of the symbol, $c$ is the compression rate, $E_{mem}$ is the energy cost to access a bit from a memory hierarchy (pJ/bit), and $E_{comp}$ is the energy cost to compress a symbol (pJ/symbol). Considering the use case of compressing DRAM data traffic, $E_{mem}$ varies depending upon the memory technologies used, for instance, from 15 pJ/bit to 47 pJ/bit [13, 55, 56] resulting in a cost up to 752 pJ/symbol to access a 16-bit symbol. We observe that the energy saving from compression is dominated by $E_{mem}$, $b$, and $c$ and not $E_{comp}$ which is over an order of magnitude lower. Note that $E_{comp}$ could be further decreased by applying more aggressive circuit level optimsations or by using different transistor technologies. If $E_{comp}$ is negligible, the Equation (2) suggests that energy saving rate from compression is exactly same as the compression rate $c$, and if not the gain will be reduced by the term $\frac{c}{b}\frac{E_{comp}}{E_{mem}}$.

## 7 EVALUATION




(a) Normalised to Lane Compression

(b) Effective compression rate normalised to the Shannon limit

Fig. 11. A comparison of compression schemes. Results are geometric means using all data sources from all networks with an error bar to show minimum and maximum.

Figure 11 compares the compression rate of Lane Compression with other methods mentioned in Section 2. Each bar shows the geometric mean with an error bar to show the minimum and maximum across all benchmark networks that we evaluated. Note that the minimum and maximum values convey slightly more meaning than just outliers. For instance, a specific compression method applied to a specific data source from a network will show a consistent compression rate as demonstrated in Section 5.2. Hence, for a system doing inference on a particular network, the performance of lossless compression could be consistently good or bad. For methods where we have observed failure to compress, this means that the method could consistently fail to compress at all. Note that we present two versions of the result for EBPC, EBPC for activations and gradients only and EBPC' for all data sources including weights. The frequent pattern table of the published EBPC [4] it not designed for weights, therefore it shows noticeable performance degradation when applied on weights. Finding the optimal frequent patterns of weights is an independent research topic and it is beyond the scope of the paper.

Figure 11(a) shows comparison in hardware cost in area of HW feasible methods normalised to the hardware cost of Lane Compression. All methods are run-time configurable and implemented internally using the same degree of circuit-level optimisations except EBPC. Hardware cost of

EBPC is retrieved from their work [4]. All methods except CSC/CSR are profiled ahead of run-time to find the best algorithmic parameters for each layer. Algorithmic parameters for CSC/CSR are borrowed from the previous implementation [6]. The DPRed allows block sizes up to 16 and S-EG supports values of the $k$ parameter up to 31. All targeted a frequency of 200MHz except EBPC which targets 600MHz.

As shown in Figure 11(a), no HW feasible method outperformed Lane Compression's geometric mean, but S-EG outperformed by 1.9% on one data source: weights for re-training from CifarNet. This data source has high sparsity but zeros are much less likely to be consecutive. Lane Compression outperformed two complex compression methods, LZW and DEFLATE, and only static Huffman coding outperformed Lane Compression by 1.3%. Static Huffman coding performed well on SqueezeNet's weights for retraining, which is the most randomly distributed data among the data sources evaluated. LZW and DEFLATE were highly effective on gradients from the LSTM, which is a text-based machine learning network. However, the overhead due to static Huffman coding was significant: the average sizes of Huffman tables were from a few kilobytes for inference to a few megabytes for re-training, *per layer*. Various additional techniques can be applied on plain static Huffman coding to reduce size of the Huffman table such as index hashing [18] or canonical Huffman coding [3], but tables cannot be removed.

In comparison to the Shannon limit, Lane Compression achieved 96.3% of the Shannon limit, outperforming all compression methods except the static Huffman coding. Moreover, with a maximum observed normalised compression rate of 139%, it is the only lightweight compression method that achieved compression rates beyond the Shannon limit. Static Huffman coding achieved 97.6% of the Shannon limit but it could never achieve beyond the limit because it is a purely symbol based method.

Figure 11(b) shows a comparison of the *effective* compression rate, where 0 means no compression and 1 means maximum symbol-based compression rate given by the Shannon limit. The effective compression rate is a 0-based compression rate and obtained by $\frac{\text{data size before compression}}{\text{data size after compression}} - 1$, instead of commonly used 1-based compression rate ($= \frac{\text{data size before compression}}{\text{data size after compression}}$) which is used elsewhere in the paper. The effective compression rate enables us to compare gains and losses from compression more directly and also highlights cases where a scheme fails to compress. Figure 11(b) shows arithmetic mean of effective compression rates normalised to the effective Shannon limit (a 0-based Shannon limit) with an error bar to show minimum and maximum. Any compression methods with error bars stretching below 0 have failed to compress in some cases. Lane Compression achieved 90.4% of the gain suggested by the Shannon limit from compression and DPRed and S-EG achieved less than 64.4% and 63.7% respectively. EBPC has been designed only for activations and gradients and struggled to compress weights, as shown by the error bar reaching near -1. CSC/CSR showed the negative average effective compression rate with the error bar reaching over -1.5. CSC/CSR is designed for sparse data therefore it was ill suited for the machine learning data without additional sparsification. Static Huffman coding shows the best effective compression rate in geometric mean (93.9%) but again never achieved beyond the limit.

Figure 12 shows compression rates of Lane Compression by network, data source, and input dataset. LeNet-5 is one of the most dense networks in values but the distribution of non-zero values is concentrated around zero, leading to a high geometric mean compression rate. On the other hand, SqueezeNet is much more sparse than LeNet-5 but non-zero values are widely distributed, making it harder to extract patterns of sequences of zeros. It also explains why Lane Compression performed less well on weights for re-training in general. Entropy density of weights are usually higher than other data sources in general and require wider bit widths to retain re-trainability, which makes distribution of non-zeros even flatter. Gradients are the most sparse data source in

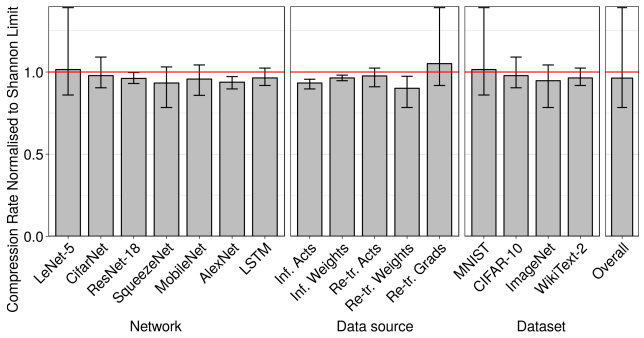



Fig. 12. Compression rate of Lane Compression normalised to the Shannon limit by network, data source, and input dataset

both values and bits, which Lane Compression favours, resulting in compression rates beyond the Shannon limit. Lane Compression could compress beyond the Shannon limit with all input datasets. Note that the lightweight compression methods specialised for zero values, ZVC, CSC/CSR, RLC, and Z-RLC, could not compress the WikiText-2 based data for all data sources except gradients.

Measured compression rates were slightly different to the estimated compression rate due to the dynamic effect of stop codes, but the difference was less than 0.19% of the estimated compression rate in terms of geometric mean. The complete set of absolute compression rates of Lane Compression can be found in Appendix A.1.

### 7.1 Interaction with Lossy Transformation

This section explores how data statistics are changed by different levels of quantization, clipping and pruning and how these changes impact the effectiveness of lossless compression techniques and the overall compression rates. We adopted the concept of parameterised intervals for quantization proposed by Jung et al. [26] which simultaneously performs both pruning and clipping, and performed a non-exhaustive exploration to find the best quantization interval for data representations with different bitwidths without additional training. This experiment is performed on activations and weights for inference from three different networks, LeNet-5, CifarNet, and MobileNet, on three different datasets, MNIST, CIFAR-10, and ImageNet respectively. For quantization, the same parameter configuration is used across all layers per data source and $\gamma$ was fixed to 1. Activations and weights were transformed and evaluated separately, unlike the data used for the rest of the paper.

Table 5 shows gradual changes in data statistics after different levels of quantization and overall results after Lane Compression is applied on the quantised data. Figure 13 shows changes in compression rates by lossy compression ($Q$, black dashed lines), lossless compression ($C$, coloured dashed lines) and also overall compression rates ($Q \times C$, solid lines) on LeNet-5, CifarNet and MobileNet. In general, value sparsity and bit sparsity increased as the resolution of data decreased but entropy density did not decrease necessarily.

As quantization level ($Q$) increases, the compression rates ($C$) of all lossless compression method showed similar tendency, and hence the relative effectiveness of different methods barely changed. Thus the consistently high compression rate of Lane Compression contributed to higher overall compression rates ($Q \times C$) compared to other methods, and the overall performance difference among different lossless compression methods became more visible as quantization level increased. We also observed that higher quantization levels ($Q$) can contribute to better compression rate

Table 5. Data statistics of different levels of lossy transformation with data type after quantization (column 'Data type ($\frac{32}{Q}$)' and resulting bits per value after applying Lane Compression on the quantised data (column 'Bits per value ($\frac{32}{Q \times C}$)').

| Network | Data source | Data type ($\frac{32}{Q}$) | Bits per value ($\frac{32}{Q \times C}$) | Value sparsity | Bit sparsity | Entropy density | Top-1 model acc. |
|---|---|---|---|---|---|---|---|
| LeNet-5 | Acts | 16 FxP | 11.8 bits | 0.02% | 58.94% | 63.76% | 99.24% (-0.00%) |
| | | 10 FxP | 6.9 bits | 1.48% | 65.14% | 64.00% | 99.22% (-0.02%) |
| | | 8 FxP | 5.5 bits | 8.43% | 66.05% | 65.47% | 99.21% (-0.03%) |
| | | 6 FxP | 4.3 bits | 7.57% | 63.85% | 72.67% | 99.06% (-0.18%) |
| | Weights | 16 FxP | 14.5 bits | 0.01% | 57.04% | 84.00% | 99.24% (-0.00%) |
| | | 10 FxP | 8.5 bits | 0.82% | 61.37% | 82.54% | 99.21% (-0.03%) |
| | | 8 FxP | 6.5 bits | 3.22% | 64.24% | 78.25% | 99.23% (-0.01%) |
| | | 6 FxP | 4.5 bits | 13.15% | 69.42% | 69.56% | 98.72% (-0.52%) |
| CifarNet | Acts | 16 FxP | 5.2 bits | 64.03% | 87.15% | 31.28% | 93.21% (-0.00%) |
| | | 10 FxP | 2.9 bits | 65.14% | 90.44% | 28.50% | 93.16% (-0.05%) |
| | | 8 FxP | 2.5 bits | 65.85% | 89.77% | 31.38% | 93.06% (-0.15%) |
| | | 6 FxP | 1.9 bits | 68.52% | 90.05% | 32.04% | 91.76% (-1.45%) |
| | Weights | 16 FxP | 13.8 bits | 0.04% | 60.13% | 84.29% | 93.19% (-0.02%) |
| | | 12 FxP | 9.8 bits | 0.59% | 63.52% | 79.79% | 93.13% (-0.08%) |
| | | 10 FxP | 7.8 bits | 2.35% | 66.28% | 75.65% | 93.14% (-0.07%) |
| | | 8 FxP | 5.8 bits | 9.42% | 70.58% | 68.73% | 92.36% (-0.85%) |
| MobileNet | Acts | 16 FxP | 6.3 bits | 54.33% | 82.43% | 39.37% | 68.15% (-0.00%) |
| | | 14 FxP | 5.4 bits | 54.74% | 83.32% | 38.84% | 68.14% (-0.01%) |
| | | 12 FxP | 4.6 bits | 54.84% | 83.35% | 40.00% | 68.01% (-0.14%) |
| | | 10 FxP | 3.8 bits | 55.15% | 84.00% | 40.44% | 67.72% (-0.43%) |
| | Weights | 16 FxP | 9.9 bits | 32.32% | 75.39% | 58.45% | 68.16% (+0.01%) |
| | | 14 FxP | 8.4 bits | 33.84% | 76.78% | 56.75% | 68.18% (+0.03%) |
| | | 12 FxP | 6.9 bits | 35.53% | 78.51% | 54.47% | 68.04% (-0.11%) |
| | | 10 FxP | 5.4 bits | 38.12% | 80.82% | 51.14% | 65.73% (-2.42%) |

($C$), as shown in Figure 13(b), which means the overall compression rates can be amplified even further in such cases. This result suggests that the data redundancy exploited by lossy and lossless compression are not of the same kind. As the level of quantization increased, the Shannon Limit, Lane Compression, and static Huffman Coding showed a similar tendency in performance. Lane Compression outperformed static Huffman coding for activations in all quantization levels, and S-EG and DPRed followed next in performance. S-EG and DPRed performed exceptionally well on weights from MobileNet compared to the weights from LeNet-5 and CifarNet, showing similar performance to Lane Compression.

These results demonstrate that the combination of lossy and lossless compression methods can achieve compression results competitive with more aggressive lossless techniques at an economic cost. Aggressive quantization methods usually involve use of expensive operations such as trigonometrical functions, multiplication, or division [12, 54, 58] which may require specialist hardware. For some techniques parameters for quantization need to be trained for each network [54, 57]

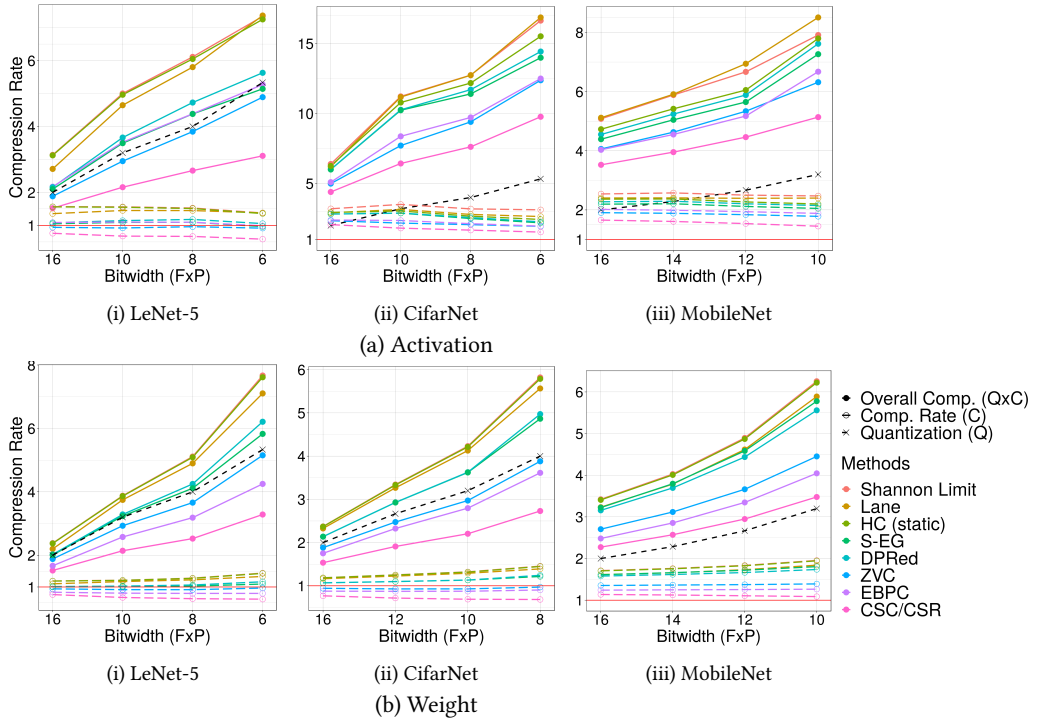



Fig. 13. Compression rates by lossless compression ($C$, coloured dashed lines), Compression rate by quantization($Q$, black dashed lines), and the overall compression rates ($Q \times C$, solid lines) by different levels of quantization on activations and weights of Lenet-5, CifarNet, and MobileNet. The overall compression rates are amplified by quatization and lossless compression.

which may not guarantee successful convergence without losing model accuracy, especially when the target rate ($Q$) is high. A lightweight lossless compression method combined with simple linear quantization can achieve fairly small bits per value as shown in column 'Bits per value ($Q \times C$)' in Table 5.

We acknowledge that the statistical characteristics might be more arbitrary as more aggressive quantization methods are developed, but such arbitrary cases were not observed on datasets with representations as narrow as 6–8 FxP. For data representations that are too narrow to be split into lanes, we can add a simple preprocessing step to restructure multiple narrow values into a value which is wide enough to split as shown in Figure 14. Such preprocessing exposes bit-wise data redundancy of the values if it exists and improves throughput by processing multiple values in a cycle without modifying the underlying hardware design of Lane Compression.

## 8 CONCLUSION

We have introduced Lane Compression, an effective and lightweight lossless compression method for machine learning. We demonstrated that on average Lane Compression outperformed all lightweight and also some complex compression methods that have been applied to machine learning data. Lane Compression is the only lightweight compression method that we observed compressing beyond the Shannon limit of the data. We have detailed a low-cost hardware implementation suitable for both on-chip and off-chip communication.

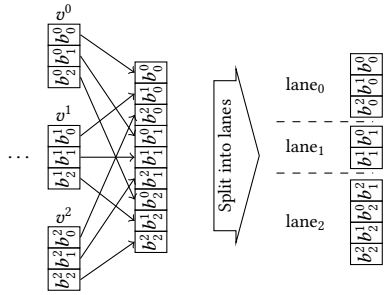



Fig. 14. A preprocessing step with an example for Lane Compression on narrow (3-FxP) values ($v^i$)

Future work will explore the effectiveness of Lane Compression on other common data formats such as BFloat16 [50] and shift weights [40]. Finding the best method for serialising machine learning data to expose the hidden data redundancy in a sequence of bits or values will be another interesting direction to investigate. Finally, additional gains are possible by considering the compressibility of data during the design of the network and during the training process [2]. We would hope that Lane Compression provides a useful parameterised compression technique to aid in this co-design process.

## ACKNOWLEDGMENTS

This work was kindly supported by the Samsung Advanced Institute of Technology (SAIT).

## REFERENCES

[1] Ziad Asghar and Jeff Gehlhaar. 2019. 2019 Snapdragon 865 5G AI Deep Dive. https://www.qualcomm.com/media/documents/files/2019-snapdragon-865-5g-ai-deep-dive-ziad-asghar-jeff-gehlhaar.pdf.

[2] Chaim Baskin, Brian Chmiel, Evgenii Zheltonozhskii, Ron Banner, Alex M. Bronstein, and Avi Mendelson. 2019. CAT: Compression-Aware Training for bandwidth reduction. arXiv:1909.11481 [cs.CV]

[3] Talal Bonny and Jörg Henkel. 2010. Huffman-based Code Compression Techniques for Embedded Processors. *ACM Trans. Des. Autom. Electron. Syst.* 15, 4, Article 31 (Oct. 2010), 37 pages.

[4] Lukas Cavigelli, Georg Rutishauser, and Luca Benini. 2019. EBPC: Extended Bit-Plane Compression for Deep Neural Network Inference and Training Accelerators. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 9, 4 (Dec 2019), 723–734.

[5] Yu-Hsin Chen, Tushar Krishna, Joel S. Emer, and Vivienne Sze. 2017. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. *IEEE Journal of Solid-State Circuits* 52, 1 (Jan 2017), 127–138.

[6] Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, and Vivienne Sze. 2019. Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 9, 2 (June 2019), 292–308.

[7] Soumith Chintala. 2016. Word-level language modeling RNN. https://github.com/pytorch/examples/tree/master/word_language_model.

[8] Yoojin Choi, Mostafa El-Khamy, and Jungwon Lee. 2020. Universal Deep Neural Network Compression. *IEEE Journal of Selected Topics in Signal Processing* (2020), 1–1.

[9] Patrick Damme, Dirk Habich, Juliana Hildebrandt, and Wolfgang Lehner. 2017. Lightweight Data Compression Algorithms: An Experimental Survey (Experiments and Analyses). In *EDBT*.

[10] Alberto Delmas, Sayeh Sharify, Patrick Judd, Milos Nikolic, and Andreas Moshovos. 2018. DPRed: Making Typical Activation Values Matter In Deep Learning Computing. *CoRR* abs/1804.06732 (2018). arXiv:1804.06732

[11] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*. 248–255.

[12] Lei Deng, Guoqi Li, Song Han, Luping Shi, and Yuan Xie. 2020. Model Compression and Hardware Acceleration for Neural Networks: A Comprehensive Survey. *Proc. IEEE* (2020), 1–48.

[13] Fabrice Devaux. 2019. The true Processing In Memory accelerator. In *2019 IEEE Hot Chips 31 Symposium (HCS)*. 1–24.

[14] Angelo Garofalo, Manuele Rusci, Francesco Conti, Davide Rossi, and Luca Benini. 2019. PULP-NN: A Computing Library for Quantized Neural Network inference at the edge on RISC-V Based Parallel Ultra Low Power Clusters. In *2019 26th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*. 33–36.

[15] Georgios Georgiadis. 2018. Accelerating Convolutional Neural Networks via Activation Map Compression. *CoRR* abs/1812.04056 (2018). arXiv:1812.04056

[16] Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C. Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. 2010. Understanding Sources of Inefficiency in General-purpose Chips. In *Proceedings of the 37th Annual International Symposium on Computer Architecture* (Saint-Malo, France). ACM, New York, NY, USA, 37–47.

[17] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. 2016. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In *Proceedings of the 43rd International Symposium on Computer Architecture* (Seoul, Republic of Korea). IEEE Press, Piscataway, NJ, USA, 243–254.

[18] Song Han, Huizi Mao, and William J. Dally. 2015. Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding. *CoRR* abs/1510.00149 (2015).

[19] Song Han, Jeff Pool, John Tran, and William J. Dally. 2015. Learning Both Weights and Connections for Efficient Neural Networks. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1* (Montreal, Canada). MIT Press, Cambridge, MA, USA, 1135–1143.

[20] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[21] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *CoRR* abs/1704.04861 (2017). arXiv:1704.04861

[22] David A. Huffman. 1952. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE* 40, 9 (Sep. 1952), 1098–1101.

[23] Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size. *CoRR* abs/1602.07360 (2016).

[24] Animesh Jain, Amar Phanishayee, Jason Mars, Lingjia Tang, and Gennady Pekhimenko. 2018. Gist: Efficient Data Encoding for Deep Neural Network Training. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 776–789.

[25] Norman P. Jouppi et al. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (Toronto, ON, Canada). ACM, New York, NY, USA.

[26] Sangil Jung, Changyong Son, Seohyung Lee, Jinwoo Son, Jae-Joon Han, Youngjun Kwak, Sung Ju Hwang, and Changkyu Choi. 2019. Learning to Quantize Deep Networks by Optimizing Quantization Intervals With Task Loss. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 4345–4354.

[27] Wonkyung Jung, Daejin Jung, , Byeongho Kim, Sunjung Lee, Wonjong Rhee, and Jung Ho Ahn. 2018. Restructuring Batch Normalization to Accelerate CNN Training. arXiv:1807.01702

[28] Hyunjun Kim. 2016. SqueezeNet v1.1. https://github.com/DeepScale/SqueezeNet/tree/master/SqueezeNet_v1.1

[29] Jungrae Kim, Michael Sullivan, Esha Choukse, and Mattan Erez. 2016. Bit-plane Compression: Transforming Data for Better Compression in Many-core Architectures. In *Proceedings of the 43rd International Symposium on Computer Architecture* (Seoul, Republic of Korea). IEEE Press, Piscataway, NJ, USA, 329–340.

[30] Morten Kjelsø, Mark Gooch, and Simon Jones. 1996. Design and performance of a main memory hardware data compressor. In *Proceedings of 22nd Euromicro Conference. Beyond 2000: Hardware and Software Design Strategies*. 423–430.

[31] Jong Hwan Ko, Duckhwan Kim, Taesik Na, Jaeha Kung, and Saibal Mukhopadhyay. 2017. Adaptive Weight Compression for Memory-efficient Neural Networks. In *Proceedings of the Conference on Design, Automation & Test in Europe* (Lausanne, Switzerland). European Design and Automation Association, 199–204.

[32] Saluka Kodituwakku and S.Amarasinghe U. 2010. COMPARISON OF LOSSLESS DATA COMPRESSION ALGORITHMS FOR TEXT DATA. *Indian Journal of Computer Science and Engineering* 1 (12 2010).

[33] Alex Krizhevsky. 2009. *Learning Multiple Layers of Features from Tiny Images*. Technical Report. University of Toronto. https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf

[34] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1* (Lake Tahoe, Nevada). Curran Associates Inc., USA, 1097–1105.

[35] Yann Lecun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (Nov 1998), 2278–2324. https://doi.org/10.1109/5.726791

[36] Arm Ltd. 2019. Arm Ethos-N series processors. https://developer.arm.com/ip-products/processors/machine-learning/arm-ethos-n.

[37] Partha Maji, Daniel Bates, Alex Chadwick, and Robert Mullins. 2017. ADaPT: Optimizing CNN Inference on IoT and Mobile Devices Using Approximately Separable 1-D Kernels. In *Proceedings of the 1st International Conference on Internet of Things and Machine Learning* (Liverpool, United Kingdom). ACM, New York, NY, USA, Article 43, 12 pages.

[38] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. 2016. Pointer Sentinel Mixture Models. *CoRR* abs/1609.07843 (2016). arXiv:1609.07843

[39] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. 2018. Mixed Precision Training. In *International Conference on Learning Representations*.

[40] Daisuke Miyashita, Edward H. Lee, and Boris Murmann. 2016. Convolutional Neural Networks using Logarithmic Data Representation. arXiv:1603.01025

[41] Miloš Nikolić, Mostafa Mahmoud, Yiren Zhao, Robert Mullins, and Andreas Moshovos. 2019. Characterizing Sources of Ineffectual Computations in Deep Learning Networks. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 165–176.

[42] Nvidia. 2019. Deep Learning Performance. https://docs.nvidia.com/deeplearning/sdk/pdf/Deep-Learning-Performance-Guide.pdf.

[43] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W. Keckler, and William J. Dally. 2017. SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (Toronto, ON, Canada). ACM, New York, NY, USA, 27–40.

[44] Richard Clark Pasco. 1976. *Source Coding Algorithms for Fast Data Compression.* Ph.D. Dissertation.

[45] Gennady Pekhimenko, Vivek Seshadri, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. 2012. Base-delta-immediate Compression: Practical Data Compression for On-chip Caches. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques* (Minneapolis, Minnesota, USA). ACM, New York, NY, USA, 377–388.

[46] Minsoo Rhu, Mike O'Connor, Niladrish Chatterjee, Jeff Pool, and Stephen W. Keckler. 2017. Compressing DMA Engine: Leveraging Activation Sparsity for Training Deep Neural Networks. *CoRR* abs/1705.01626 (2017). arXiv:1705.01626

[47] Mark A. Roth and Scott J. Van Horn. 1993. Database Compression. *SIGMOD Rec.* 22, 3 (Sept. 1993), 31–39.

[48] Amir Said. 2004. *Introduction to Arithmetic Coding - Theory and Practice.* Technical Report HPL-2004-76. Imaging Systems Laboratory, HP Laboratories Palo Alto.

[49] Claude E. Shannon. 1948. A Mathematical Theory of Communication. *Bell System Technical Journal* 27, 3 (1948), 379–423.

[50] Giuseppe Tagliavini, Stefan Mach, Davide Rossi, Andrea Marongiu, and Luca Benini. 2018. A transprecision floating-point platform for ultra-low power computing. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*. 1051–1056.

[51] Richard Wilson Vuduc. 2003. *Automatic Performance Tuning of Sparse Matrix Kernels.* Ph.D. Dissertation. AAI3121741.

[52] Ying Wang, Huawei Li, and Xiaowei Li. 2018. A Case of On-Chip Memory Subsystem Design for Low-Power CNN Accelerators. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 10 (Oct 2018), 1971–1984.

[53] Terry A. Welch. 1984. A Technique for High-Performance Data Compression. *Computer* 17, 6 (June 1984), 8–19.

[54] Jiwei Yang, Xu Shen, Jun Xing, Xinmei Tian, Houqiang Li, Bing Deng, Jianqiang Huang, and Xian-sheng Hua. 2019. Quantization Networks. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[55] Xuan Yang, Mingyu Gao, Qiaoyi Liu, Jeff Setter, Jing Pu, Ankita Nayak, Steven Bell, Kaidi Cao, Heonjae Ha, Priyanka Raina, Christos Kozyrakis, and Mark Horowitz. 2020. Interstellar: Using Halide's Scheduling Language to Analyze DNN Accelerators. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 369–383. https://doi.org/10.1145/3373376.3378514

[56] Amir Yazdanbakhsh, Choungki Song, Jacob Sacks, Pejman Lotfi-Kamran, Hadi Esmaeilzadeh, and Nam Sung Kim. 2018. In-DRAM near-Data Approximate Acceleration for GPUs. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques* (Limassol, Cyprus) *(PACT '18)*. Association for Computing Machinery, New York, NY, USA, Article 34, 14 pages. https://doi.org/10.1145/3243176.3243188

[57] Dongqing Zhang, Jiaolong Yang, Dongqiangzi Ye, and Gang Hua. 2018. LQ-Nets: Learned Quantization for Highly Accurate and Compact Deep Neural Networks. In *Computer Vision – ECCV 2018*, Vittorio Ferrari, Martial Hebert, Cristian Sminchisescu, and Yair Weiss (Eds.). Springer International Publishing, Cham, 373–390.

[58] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. 2016. DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients. arXiv:1606.06160

# A APPENDIX

## A.1 A complete set of absolute compression rate of Lane Compression

Table 6. Absolute compression rate of Lane Compression against quantised data compared to the Shannon limit (*Limit*), estimated compression rates from profiling (*Est. (SW)*), measured from the hardware implementation (*Meas. (HW)*), and overall compression rates against 32-bit full precision data (*vs. 32-bit*).

| Network | Inf./ Re-tr. | Data Source | Limit ($C$) | Est. (SW, $C$) | Meas. (HW, $C$) | vs. 32-bit ($Q \times C$) |
|---|---|---|---|---|---|---|
| LeNet-5 | Inf. | Acts | 1.56 | 1.48 | 1.47 | 4.70 |
| | | Weights | 1.21 | 1.17 | 1.17 | 3.75 |
| | Re-tr. | Acts | 1.35 | 1.34 | 1.34 | 2.68 |
| | | Weights | 1.42 | 1.22 | 1.22 | 2.44 |
| | | Grads | 1.77 | 2.44 | 2.46 | 4.92 |
| CifarNet | Inf. | Acts | 3.39 | 3.06 | 3.06 | 10.88 |
| | | Weights | 1.32 | 1.29 | 1.29 | 4.13 |
| | Re-tr. | Acts | 1.64 | 1.61 | 1.61 | 3.30 |
| | | Weights | 1.80 | 1.71 | 1.70 | 3.40 |
| | | Grads | 1.90 | 2.07 | 2.08 | 4.16 |
| ResNet-18 | Inf. | Acts | 1.98 | 1.87 | 1.85 | 5.32 |
| | | Weights | 1.39 | 1.36 | 1.36 | 3.63 |
| | Re-tr. | Acts | 1.43 | 1.39 | 1.39 | 1.88 |
| | | Weights | 1.28 | 1.19 | 1.19 | 1.59 |
| | | Grads | 1.83 | 1.82 | 1.83 | 2.44 |
| SqueezeNet | Inf. | Acts | 2.11 | 2.01 | 2.01 | 5.81 |
| | | Weights | 1.40 | 1.35 | 1.33 | 3.56 |
| | Re-tr. | Acts | 1.45 | 1.41 | 1.40 | 1.90 |
| | | Weights | 1.54 | 1.21 | 1.21 | 1.61 |
| | | Grads | 2.15 | 2.24 | 2.21 | 2.95 |
| MobileNet | Inf. | Acts | 2.57 | 2.45 | 2.46 | 6.03 |
| | | Weights | 1.76 | 1.67 | 1.67 | 3.81 |
| | Re-tr. | Acts | 1.89 | 1.88 | 1.87 | 2.53 |
| | | Weights | 1.76 | 1.51 | 1.51 | 2.01 |
| | | Grads | 1.98 | 2.07 | 2.06 | 2.75 |
| AlexNet | Inf. | Acts | 2.34 | 2.11 | 2.10 | 6.00 |
| | | Weights | 1.35 | 1.33 | 1.30 | 3.48 |
| | Re-tr. | Acts | 1.59 | 1.45 | 1.45 | 1.95 |
| | | Weights | 1.16 | 1.13 | 1.13 | 1.51 |
| | | Grads | 2.63 | 2.51 | 2.50 | 3.33 |
| LSTM | Inf. | Acts | 1.66 | 1.57 | 1.57 | 4.19 |
| | | Weights | 1.82 | 1.75 | 1.75 | 4.66 |
| | Re-tr. | Acts | 1.34 | 1.32 | 1.37 | 2.19 |
| | | Weights | 1.41 | 1.38 | 1.38 | 2.20 |
| | | Grads | 4.90 | 4.50 | 4.50 | 7.20 |