Atalanta: A Bit is Worth a "Thousand" Tensor Values

Alberto Delmas Lascorz
a.delmaslascorz@mail.utoronto.ca
University of Toronto
Toronto, Canada

Mostafa Mahmoud
mostafa.mahmoud@mail.utoronto.ca
University of Toronto
Toronto, Canada

Ali Hadi Zadeh
a.hadizadeh@mail.utoronto.ca
1QBit Ltd & University of Toronto
Toronto, Canada

Miloš Nikolić
milos.nikolic@mail.utoronto.ca
University of Toronto
Toronto, Canada

Kareem Ibrahim
kareem.ibrahim@mail.utoronto.ca
University of Toronto
Toronto, Canada

Christina Giannoula
christina.giann@gmail.com
University of Toronto
Toronto, Canada

Ameer Abdelhadi
ameer@mcmaster.ca
McMaster University
Hamilton, Canada

Andreas Moshovos
moshovos@ece.utoronto.ca
University of Toronto & Vector Institute
Toronto, Canada

Abstract

Atalanta is a *lossless*, hardware/software co-designed compression technique for the tensors of *fixed-point* quantized deep neural networks. *Atalanta* increases effective memory capacity, reduces off-die traffic, and/or helps to achieve the desired performance/energy targets while using smaller off-die memories during inference. *Atalanta* is architected to deliver nearly identical coding efficiency compared to Arithmetic Coding while avoiding its complexity, overhead, and bandwidth limitations. Indicatively, the *Atalanta* decoder and encoder units each use less than 50B of internal storage. In hardware, *Atalanta* is implemented as an *assist* over any machine learning accelerator transparently compressing/decompressing tensors just before the off-die memory controller. This work shows the performance and energy efficiency of *Atalanta* when implemented in a 65nm technology node. *Atalanta* reduces data footprint of weights and activations to 60% and 48% respectively on average over a wide set of 8-bit *quantized* models and complements a wide range of quantization methods. Integrated with a Tensorcore-based accelerator, *Atalanta* boosts the speedup and energy efficiency to 1.44× and 1.37×, respectively. *Atalanta* is effective at compressing the stashed activations during training for fixed-point inference.

ASPLOS '24, April 27-May 1, 2024, La Jolla, CA, USA

ACM ISBN 979-8-4007-0385-0/24/04
https://doi.org/10.1145/3620665.3640356

ACM Reference Format:
Alberto Delmas Lascorz, Mostafa Mahmoud, Ali Hadi Zadeh, Miloš Nikolić, Kareem Ibrahim, Christina Giannoula, Ameer Abdelhadi, and Andreas Moshovos. 2024. Atalanta: A Bit is Worth a "Thousand" Tensor Values . In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3620665.3640356

1 Introduction

The memory footprint of state-of-the-art deep learning (DL) models (hitherto referred to as *models*) has been rapidly increasing. Designers use more layers and weight parameters to improve task performance, tackle increasingly more complex tasks, and capture more complex patterns in the deluge of available data to train on. As a result, the energy efficiency and execution performance of state-of-the-art models remains limited by accesses to tensor data from off-die memory and in some cases across network links [21, 34].

Hardware-based memory value compression directly reduces these off-die access costs. It has been extensively studied for general purpose systems [17, 63, 72, 84] where it capitalizes on program-related value behavior such as common address prefixes, and other repetitive patterns. Unfortunately, these compressors are not well-suited for addressing the unique properties and requirements of DL [48]. First, DL tensors exhibit Gaussian-like distributions with a few outlier values and practically none of the aforementioned patterns [48, 71]. Second, altering model values is tolerable as long as task performance metrics, like accuracy, are preserved. Third, DL models predominantly require wide streaming accesses rather than narrow, random accesses.

The different opportunities and needs of deep learning have not gone unnoticed. DL-specific *quantization* [27, 51, 53, 71] and value *compression* [13, 14, 25, 29, 30, 43, 44, 48, 54, 60, 77] are widely adopted to reduce memory traffic.

Quantization capitalizes on the tolerance for value changes and trains or modifies the model to use a sparser value space that requires fewer bits per value [10, 16, 27, 30, 51, 67, 71, 99, 100]. Today, quantization to 8 bit fixed-point is typically possible with little or no effect on task performance and for many tasks. Quantization to 4b has been demonstrated for inner layers [16] whereas more extreme quantization affects accuracy much more [15, 16, 19]. Typically quantization uses a fixed data width among all values of the same tensor, layer, or network. Given that typically only a small number of values require the maximum data width needed for the tensor (or network) as a whole, these methods result in lost encoding efficiency opportunities. Sections 2 and 5.8 further consider the interplay between quantization and compression.

Prior works in hardware data compression for DL models [13, 14, 29, 30, 44, 53, 54, 77], as we also show in Section 5, are still far from a near-optimal encoding. Specifically, these works (i) either compress only a subset of the values, e.g., only zeros [14, 29, 53, 54, 77], only the weight parameters [30], or (ii) use a fixed data width (e.g., 5-bit) per value [60] or groups of data values [48], or (iii) target specific, expected value patterns, e.g., [13, 44].Compression methods like those from the LZ77 family [22, 94, 101], which aren't specific to DL models, have proven effective in applications where their significant hardware demands, including local storage, frequent lookups, and encoding table updates, are acceptable. We consider such methods in Section 5.8.

We propose *Atalanta*, a near-optimal, practical, and lossless hardware/software compressor for efficient DL inference with application also in training (see Section 5.10). *Atalanta* meets the following requirements: First, it provides near-optimal encoding of weights *and* activations (e.g., it can encode frequent values with *less than a bit*). Second, it has low area and power costs to be seamlessly integrated with systems tailored for executing DL models, as well as domain-specific accelerators such as Tensorcore units [24, 40], systolic arrays [42] and sparsity-specific DL accelerators [29, 70]. Third, *Atalanta* provides transparent and lossless compression, leaving task performance intact. Fourth, it is general enough to support any arbitrary (quantized or not) model using fixed-point value tensors. Fifth, *Atalanta* naturally adapts and thus rewards quantization to fewer bits where and when this is possible without necessitating it.

Atalanta is inspired by Arithmetic Coding (AC) [8, 78, 79], which achieves nearly *optimal* encoding efficiency by using a real number in the range $[0, 1)$ to represent an input symbol sequence (here tensor values). The precision (the number of bits) required for encoding depends on the sequence and the expected frequency of values. Uniquely, AC can encode frequent values using, effectively, less than one bit.

Using AC to enhance DL application energy efficiency is an open challenge due to AC's high implementation and power costs (see Sections 3.1 and 5). AC has been primarily used in scenarios where communication costs overwhelmingly exceed that of compression, making maximum compression the ultimate objective (e.g., transmitting video over low-bandwidth links or storing it on hard drives). Architecting an AC-based solution that provides optimal encoding and low hardware costs (including power, latency, bandwidth, and area) is challenging for several reasons. 1) To maximize the compression ratio, it is necessary to keep track of the expected frequency for *each* possible value, which leads to prohibitively large area and power costs for table lookups. 2) AC is inherently sequential, producing only a few bits per decoding step, which conflicts with the wide and high-bandwidth needs of DL. 3) AC requires costly arithmetic operations, such as wide multiplications and divisions, which further increase energy, latency, and area costs in hardware. 4) AC requires *advance knowledge* of the value distribution, which is challenging for activations.

Atalanta offers near-AC encoding efficiency and practicality for seamless integration with deep learning (DL) accelerators at a low cost, achieved through the following key techniques: 1) Partitioning the value space per tensor into sub-ranges allows each value v to be mapped to a ($symbol$, $offset$) pair, where $v = symbol + offset$, with values within the same sub-range sharing a common base $symbol$. This approach enables *Atalanta* to track probabilities at a sub-range granularity, eliminating the high area and power costs of AC hardware, with minimal loss in compression effectiveness (Section 5). Customizing sub-ranges for each tensor is achieved through a heuristic algorithm that maximizes compression effectiveness (Section 4.3). 2) To meet the high-bandwidth requirements of DL, *Atalanta* divides tensors into chunks and integrates multiple hardware coder/encoder units, increasing execution parallelism. Each unit independently processes a chunk of the tensor, a common approach in DL accelerators [39], producing a single value per cycle. 3) *Atalanta* eliminates expensive AC arithmetic operations by approximating them with lightweight calculations, using only shift operations and narrow fixed-point multiplications and additions. 4) The key observation that the distribution of activation values remains almost the same regardless of input allows *Atalanta* to pre-select sub-ranges and their probabilities for activations through profiling.

We evaluate *Atalanta* using a wide variety of DL models, and highlight the following experimental results:

- Implemented in a 65nm tech node, *Atalanta* encoder and decoder occupy *only* $0.02mm^2$ and $0.017mm^2$ area, respectively, and consume $2.8mW$ and $2.65mW$ power, respectively (this is measured over a subset of the models).
- Per model, the compression rate for activations varies from 1.43× to 4.2×, and from 1.13× to 11.4× for weights.
- *Atalanta* rewards quantization and pruning delivering reductions in off-die traffic.

- When integrated with a Tensorcore-based accelerator, *Atalanta* improves performance and energy efficiency by 1.44× and 1.37×, respectively.
- *Atalanta* rivals and often reduces footprint more than the much more expensive Deflate method.

2 To Quantize and Compress

Deep learning models are not set in stone. Their architecture, value representation, and arithmetic can change as long as they perform well on their target tasks. However, designing and training these models is challenging. After selecting an architecture, it's usually trained using floating-point values and carefully selected or painstakingly explored hyperparameters and recipes. Once effective, designers often streamline the model using *quantization*.

Quantization alters the value representation and arithmetic used in inference. A direct alteration usually hurts task performance which can often recover by retraining or fine-tuning the model by running extra training epochs. Ideally, retraining would always be feasible and cost-effective. However, it demands access to the training dataset, ample computational resources, time, potential changes to hyperparameters, and ensuring the final task performance loss is tolerable. Meeting all these conditions isn't always feasible. Regardless, there are many quantized models that proved sufficient and efficient (Section 5 studies such models).

The body of quantization work far exceeds what can be reasonably covered here, e.g., [27, 51]. We study the interplay of quantization and compression using representative methods from two primary classes: a) direct quantization to narrow-fixed point values, and b) indirect quantization to a dictionary of representative values.

Direct Quantization: These methods use fixed-point integers, often with a few bits. Quantization to 8-bit integers is commonly effective for vision tasks and frequently results in tolerable task performance loss in others, e.g., [4, 45]. Narrower bitwidths have also been demonstrated e.g., [15, 16, 20, 67]. However, the chosen bitwidth must accommodate all values, including rare high-magnitude outliers. This can be inefficient for skewed distributions where most values are small. *Atalanta* excels at exploiting such distributions and benefits from pruning [53]. Section 5 shows the effectiveness of our method in models using 8-bit to 2-bit per tensor precisions for ResNet18 [67] and 4-bit models [16]. An additional advantage of *Atalanta* is its flexibility to support any bit length, thus enhancing any future advances in direct quantization.

Indirect (Dictionary) Quantization: These methods quantize values to a few representative values, or *centroids*, e.g.,[30]. They reduce footprint for two reasons: 1) they store values as narrow indexes to the dictionary of centroids, and 2) they can spread the centroid selection in the target value range to better match the expected distribution. We defer the discussion and evaluation of the interplay of Atalanta and Indirect Quantization until Section 5.12.

3 Arithmetic Coding Primer

This section is a refresher of arithmetic coding fundamentals. Reflecting on our own experience, arithmetic coding requires a non-trivial investment to comprehend. We refer the reader to [46, 65] for more detailed introductions.

Informally, Arithmetic Coding (AC) converts a sequence of symbols (such as tensor values) into a *code*, a real number in $[0, 1)$, which can be used to reproduce the input sequence. The code's precision (number of bits) is determined by the order of symbols and their expected frequency, making it potentially large. The encoding process involves adjusting a value range that encodes the sequence of symbols seen so far. The code can be any value within this range. As symbols are processed, the range narrows, with the relative position of the code encoding the sequence seen so far. The less frequent the symbol, the narrower the range, requiring more bits for precision. To achieve maximal compression, Arithmetic Coding requires advance knowledge of the symbol frequency and arbitrary precision and arithmetic.

Let us consider an example of encoding four symbols, `A` through `D`, respectively with frequencies of 0.4, 0.1, 0.3, 0.2. AC could assign range $[0, 0.4)$ to `A`, $[0.4, 0.5)$ to `B`, and $[0.5, 0.8)$ and $[0.8, 1.0)$ respectively to `C` and `D`. A *single* B can be represented by any number in $[0.4, 0.5)$. While this may seem inefficient, it is only so because presently we are looking at single values. Figure 1 shows how the sequence `ABA` ends up being encoded with a code in $[0.16, 0.176)$, e.g., 0.16. As it encounters the symbols, coding progressively adjusts the position and size of the initial $[(low) = 0.0, (high) = 1.0)$ range. For example, encoding A restricts the range to $[0.0, 0.4)$. Encoding the B that follows, further restricts the range to its $[0.4, 0.5)$ sub-range which is $[0.16, 0.20)$.

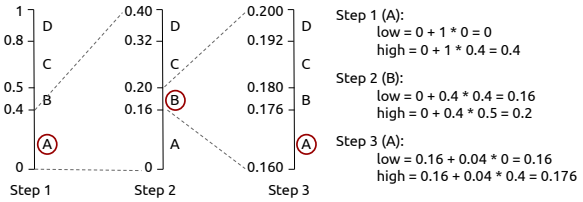

Figure 1. An example of AC encoding.

Formally, AC accepts a sequence S of input symbols $S = s_N ... s_i ... s_0$ from a vocabulary V of symbols $\{v_s, ..., v_0\} \in V$ and a table of ranges/probabilities $\left[high_j, plow_j\right)$, one per symbol in V. For maximal compression, each range is sized proportional to the probability of occurrence of the respective symbol. The result is a *code* value, a number in $[0, 1)$ which uniquely represents the input sequence S. Internally, the method uses two state variables *high* and *low*. The following is a pseudo-code implementation:

Encoding starts with a $[0.0, 1.0)$ range (#1). Each symbol s_i is read (#3) and used to index the table of ranges (#4). Line #5 calculates the current *range* length. The new boundaries are

Algorithm 1: Infinite precision Arithmetic Coding (AC).

Data: $S = s_N, ..., s_0, phigh_s, ..., phigh_0, plow_s, ..., plow_0$
Result: *code* representing the input sequence S
1 $low = 0.0$; $high = 1.0$;
2 **while** $i < N$ **do**
3 $s = s_i$
4 $probh = phigh_{s_i}$; $probl = plow_{s_i}$
5 $range = high - low$
6 $high = low + probh \times range$
7 $low = low + probl \times range$
8 i++
9 **end**
10 $code \leftarrow low$

offset from the current low by adding the scaled with *range* symbol boundaries (#6-#7). Once the full symbol sequence has been processed, the code can be any value between low and high. Decoding performs the inverse computation.

3.1 Challenges with Arithmetic Coding

The above arithmetic coding method is not a good fit for our purposes: 1) It requires expandable precision arithmetic, 2) uses a range table with one entry per potential input value, and 3) performs multiple expensive operations (e.g., decoding would require arbitrary precision division).

Expandable precision arithmetic implementations are costly and unsuitable for our needs, as execution time grows proportionally with precision, reaching thousands of cycles for typical tensors. This makes them unsuitable for our purposes. Fortunately, there are AC approximations that use finite-precision arithmetic.

Atalanta's AC component is inspired by Nelson's software AC implementation [65]. Readers can refer to Nelson's explanation of this non-trivial modification to AC. Due to space limitations, it suffices to note that Nelson's implementation, while less costly than pure AC, it is still impractical for our purposes due to 1) its requirement of a probability entry per value, which necessitates a 256-entry table for 8b models, 2) severe bandwidth limitations as it encodes and decodes a single bit at a time, and 3) the use of expensive operations proportional to the number of probability table entries.

Even if we assume that we can alter Nelson's implementation to process a whole value per step and to avoid the cost of expensive operations, just the cost of the probability table proves prohibitive. A properly configured coder/decoder unit for 8b models, would need to have a table of $256(symbols) \times 10b(probability) \times 2 \approx 5Kb$ of storage just for the probability table (with higher overheads for 16b models which are still in use in applications requiring high resolution output, such as segmentation). Since we have to replicate the units to achieve high bandwidth data supply, these costs are multiplied further. The resulting power for the practical configuration studied in Section 5.6 proves prohibitive reaching an overhead of 93.6% vs. DRAM power alone.

4 Atalanta

Atalanta introduces a modified encoding/decoding process where 1) all updates to the state (high, low, and code) are performed in a single step, and 2) arithmetic coding is used only for a variable portion of each value. As a result, it uses tables with at *most* 16 entries and low complexity operations (shifts and narrow fixed point multiplication), resulting in a novel energy efficient and higher bandwidth implementation. *Atalanta* manages to use very few entries by partitioning the input value space into several non-overlapping ranges $[v_{min}, v_{max}]$ with *no* constraints on range spread or origin. A value v is encoded as $(symbol, offset)$ where $symbol = v_{min}$ and $offset = v - symbol$, and we define an unsigned integer $OL = log_2(v_{max} - v_{min})$ bits, that represents the bitwidth of the range of the values. We have experimented with several 4b, 8b, and 16b models and found that using 16 ranges, with 8b v_{min} and v_{max}, and 3b OL (4b for 16b models) is sufficient (see Section 5.6 — more than 16 ranges at best improved compression by 0.1% or less, whereas for many models even 8 ranges would have been enough). While *Atalanta*'s implementation supports 16 ranges, by appropriately setting the range boundaries, we do not have to use them all further improving energy efficiency.

At a high-level, *Atalanta* innovates over AC as follows:

- AC requires the expected frequency per possible value, incurring prohibitive area and power costs for table lookups. Instead, *Atalanta* partitions the value space per tensor into sub-ranges, and tracks of the expected frequency of each sub-range, thus providing low area/power costs.
- The original AC is inherently sequential, producing only a few bits per decoding step. Instead, *Atalanta*: 1) groups all per bit operations into a single bit-parallel operation per input value (encode) and per symbol (decode), and 2) splits tensors in chunks and integrates multiple hardware units to increase execution parallelism.
- AC requires costly (energy, latency, and area) arithmetic operations, such as wide multiplications and divisions. In contrast, *Atalanta* uses only shift operations and narrow fixed-point multiplications and additions, and thus achieving low energy/latency/area costs.
- AC requires advance knowledge of the distribution of all values. Instead, *Atalanta* needs to know the data value distribution at a sub-range granularity, which is achieved via lightweight profiling. This also makes *Atalanta* more tolerant in variations in the input distribution across inputs.
- *Atalanta* uses profiling and a heuristic algorithm to split the input value range into subranges optimizing the footprint of the encoded stream.

We detail *Atalanta* data value encoding and decoding.

Encoding: The *Atalanta* encoder compresses a value sequence into two streams. It first maps each value into a (*symbol*, *offset*) pair and then arithmetically encodes just the *symbol* (v_{min}) (1st stream) while storing the *offset* verbatim using only as many bits as necessary (2nd stream). Very frequent symbols may end up using no *offset* bits. *Atalanta* also stores the following metadata: 1) the number of symbols encoded, and 2) the range and probability table (encoding table for short) which needs less than 50 bytes for our example implementation.The table encodes the value range (v_{min}, v_{max}) and the assigned probability range ($low_i, high_i$) per symbol. We use 10b for the probabilities, and we assign the full range of ($0x0$, *0x3ff*) across the symbols. A software optimizer (Section 4.3), determines per tensor value ranges that reduce the overall footprint.

Table 1 shows an example encoding table for the weights of a layer of BILSTM (see Section 5). The row index "IDX" and the symbol probability 'p' are shown for clarity — they are not stored. Row 0 assigns the four values in $[0x00, 0x03]$ the $[0x000, 0x1EB)$ range (a probability 'p' of 0.4795). Any of these values will be encoded as symbol 0 and will be decoded to $v_{min} = 0x00$. The exact value v is recovered by adding the recorded $OL = 2$b to v_{min}. Values absent from the tensor are assigned a zero probability with $tlow = thigh$ (rows 3-12). Row 13 captures all values in $[0xD0, 0xF3]$ assigning them symbol 13. Notice that the offset requires 6 bits since $0xF0 - 0xD0 = 0x23$. Since $0x23 < 2^6 - 1$ not all offset values will be used. Our implementation store the symbols in order such that $v_{min}[i] = v_{max}[i-1] + 1$ for $i > 0$ allowing us to store only v_{max} and $thigh$ per row reducing costs.

Table 1. Symbol and Probability Count Table Example

IDX	v_min	v_max	OL	tlow	thigh	p
0	0x00	0x03	2	0x000	0x1EB	0.4795
1	0x04	0x07	2	0x1EB	0x229	0.0605
2	0x08	0x0F	3	0x229	0x238	0.0146
3	0x10	0x3F	6	0x238	0x23A	0.0020
4	0x40	0x4F	4	0x23A	0x23A	0.0000
5	0x50	0x5F	4	0x23A	0x23A	0.0000
6	0x60	0x6F	4	0x23A	0x23A	0.0000
7	0x70	0x7F	4	0x23A	0x23A	0.0000
8	0x80	0x8F	4	0x23A	0x23A	0.0000
9	0x90	0x9F	4	0x23A	0x23A	0.0000
10	0xA0	0xAF	4	0x23A	0x23A	0.0000
11	0xB0	0xBF	4	0x23A	0x23A	0.0000
12	0xC0	0xCF	4	0x23A	0x23A	0.0000
13	0xD0	0xF3	6	0x23A	0x23C	0.0020
14	0xF4	0xFB	3	0x23C	0x276	0.0566
15	0xFC	0xFF	2	0x276	0x3FF	0.3838

Listing 1 shows in pseudo-code the encoding process. The 16b (HIGH,LOW) registers hold the current range (initially (1.0, 0] = (0xffff, 0x0000)). A 5b underflow bit counter UBC avoids cases where 16b arithmetic is insufficient. Encoding a value proceeds as follows: **Step 1:** Lookup which entry of the probability table the 8b input 'val' matches (using the v_{min}, v_{max} fields (lines 8-10). **Step 2:** Using the OL field, extract the corresponding least significant bits from 'val' and write them to the offset stream (lines 11-12). **Step 3**: Adjust (HIGH, LOW) according to the value's probability (lines 13-16). **Step 4:** Shift out to the Symbol stream any common prefix of (HIGH, LOW) (lines 17-23) filling their least significant bit positions with 1s and 0s respectively. If there have been underflow bits detected earlier (non-zero UBC), place them after the first output bit, setting them to its inverse value (see Section 4.1). **Step 5:** Prevent cases where 16b arithmetic is not enough (lines 27-31). Ignoring the most significant bit of (HIGH, LOW), check whether there is a prefix of 0s in HIGH that is matched with a prefix of 1s in LOW. Shift those out counting the bit length in UBC. Table 2 shows an encoding example.

Table 2. Encoding the value sequence 0xff, 0x03 using the example probability count table of Table 1.

Initial State:	(HIGH,LOW) = (0xffff, 0x0000) implies range = 0x10000
Input Value	0xff
Lookup PCNT [v_min, v_max]:	entry 15 (tlow, thigh, OL)=(0x276, 0x3ff, 2)
Output to Offset Stream	0xff >>OL $->11_{(2)}$
Shrink (HIGH, LOW):	HIGH = 0x0000 + (0x10000 * 0x3ff) >>10 -1 = 0xffbf LOW = 0x0000 + (0x10000 * 0x276) >>10 = 0x9d80
(HIGH,LOW) common prefix?	1b long, write it to AC stream Output a $1_{(2)}$
Adjust (HIGH,LOW)	shift left filling HIGH with 1s and LOW with 0s (HIGH,LOW) = (0xff7f, 0x3b00)
Input Value	0x3
Lookup PCNT [v_min, v_max]:	entry 0 (tlow, thigh, OL) = (0x0, 0x1eb, 2)
Output to Offset Stream	0x3 >>OL $->11_{(2)}$
Shrink (HIGH, LOW):	HIGH = 0x3b00 + (0x1eb * 0xc480) >>10 - 1 = 0x9937 LOW = 0x3b00 + (0x0 * 0xc480) >>10 = 0x3b00
(HIGH, LOW) common prefix?	No

Decoding: The decoder (pseudo-code in Listing 2) accepts as input two sequences: 1) the compressed symbols and 2) the corresponding offsets, and outputs the original values. At each step, it first decodes a value prefix from the symbol sequence. Using the symbol table, it then extracts the appropriate number of offset bits, which it adds to the value prefix. The process continues until all symbols have been decoded. Our decoder produces a single value per step using a similar process to the encoder. It maintains (HIGH, LOW) registers and a 16b CODE register which is a window into the symbol stream. At each step, to decode a symbol, it scales CODE by the current range (HIGH,LOW), offsets it by LOW, and then scans the probability range table to find the matching range.

4.1 Implementation

Encoder: Figure 2 shows the encoder implementation which mirrors Listing 1. We provide an overview emphasizing the handling of over/underflow. Separate encoders for activations and weights are initialized *once* per layer: 1) HI_in and LO_in: set the 16b HIGH and LOW (the extra 1b is an enable). 2) SYMT_in: sets the symbol table (v_{min} and offsets) once per tensor. The implementation assumes that: i) the full range is mapped, and ii) the rows are ordered in value (symbol) order.

```
1  HIGH, LOW: 16b regs, initially 0xffff, 0x0000
2  IN: input value
3  PCNT[]: Symbol & Prob. Count Table (Table 1)
4  val: 8b input value
5  UBC: 5b outstanding underflow bits, initially 0
6
7   Encode_Value:
8    Table Lookup:
9      find i s.t. PCNT[i].vmin≤ val ≤PCNT[i].vmax
10         tlow,thigh,OL = PCNT[i].(tlow,thigh,OL)
11   Output to Offset stream:
12   write the OL LSbs of val-PCNT[i].vmin
13   Adjust high & low:
14   range = HIGH - LOW + 1
15   HIGH = LOW + (range * thigh) >> 10 - 1
16   LOW = LOW + (range * tlow) >> 10
17   Output to Arithmetic Coding Stream:
18   HIGH & LOW common prefix?
19     Shift out to Symbol stream:
20     if UBC != 0 // underflow
21       shift out MSb from HIGH and LOW
22       output the inverse of MSb UBC times
23       UBC = 0
24
25     vacated LSbs: fill w/ 1s/0s for HIGH/LOW
26   Underflow Handling:
27     (HIGH,LOW) of the form x0...0xxx,x1...1xxx)?
28     UBC += #bits in (0...0)
29     remove 0...0 and 1...0 from (HIGH,LOW)
30     vacated LSbs: fill w/ 1s/0s for HIGH/LOW
```

Listing 1. Atalanta encoding a value

3) PCTN_in: sets the probability count entries (thigh,tlow) 10b each (ordered to match the symbol table). The encoder produces the following outputs: 1) OFS_out: an up to 8*b* value containing the corresponding offset and its 4b length OFS_r (range 0b-8b). 2) A 16*b* CODE_out contains CODE_c (4b+1b to indicate no output) useful bits to be written into the encoded symbol stream. The 5*b* CODE_u signals how many additional "underflow" bits (explained below) to output. The bits are inserted after the most significant bit of CODE_out and they are set to its inverse.

Every step the encoder receives a value via the 8*b* IN port (1*b* enable). The 1*b* "done" signal terminates encoding. The encoder's internal state is kept in three registers: 5b UBC, and 16b HI and LO. Encoding starts in "SYMBOL Lookup" by finding which range the incoming value (SYMT_in) corresponds to. The range index (SYM[i]) is used: 1) to extract the appropriate least significant bits from the value and to output them to the OFFSET stream (OFS_out (bits) and OFS_r (how many)) and 2) in parallel, to lookup the probability boundaries of the range ("PCNT Table"), and then ("Hi/Lo/CODE Gen") to scale the HI and LO range boundaries, and finally to output to the symbol stream any common prefix in them: CODE_out (bits) and CODE_c (how many bits).

Underflow Detection and Handling: The encoder uses finite precision arithmetic to execute an algorithm that requires arbitrary precision arithmetic. It effectively maintains a window of 16b into what are high and low range boundaries of arbitrary bit precision. The HI and LO registers contain this 16b-wide window and conceptually, they have suffixes of an "infinite" number of 1s and 0s respectively. The window is allowed to slide to less significant bits by shifting out any prefix that can no longer change. As we encode one symbol after the other, the HI and LO boundaries shrink with HI always becoming smaller and LO growing larger. However, HI>LO always holds since each encoded symbol is of non-zero probability. As HI and LO approach they will grow an increasingly longer common prefix. Those are the bits that the encoder can safely "discard" by shifting them out of the HI and LO register writing them on the encoded stream.

However, there are cases, where depending on the probability range of a new symbol and the current range, having a window of just 16b is not enough to appropriately scale the range so that HI remains larger than LO. The case is where HI contains a value of the form 100... and LO a value of the form 011... which means that HI and LO are converging around 0.5 (as viewed within the current window and thus ignoring the common prefix bits that have been shifted out). To eventually find out whether they will end up being both above 0.5 or below it, requires arithmetic with more than 16b. This happens when the range adjustments done are so small that they need to affect bits that are not yet within the current window. The encoder handles such cases *preemptively* by entering a state where it records how many *underflow* bits are needed allowing the window to slide (lines 26-30 in Listing 1). The encoder handles this by identifying, starting from the second most significant bit, any prefix of tHI' and tLO' where tLO is all 1s and tHi is all 0s. This subprefix is shifted away from tHI' and tLO' yielding tHI" and tLO". To detect the length of this subprefix, the encoder uses a leading 1 detector for tHI' (ignoring the MSb – most significant bit) and a leading 0 for tLO' (again ignoring the MSb). The subprefix is the most significant position among the two. This is implemented in the 01PREFIX block. This subprefix is removed from tLO' and tHI' producing tLO" and tHI". Its length is added to the UBC register which counts the number of outstanding underflow bits. Those eventually we will be set to the inverse of the most significant bit of HI (1s or 0s if we end up respectively on the upper or lower part of the range below or above 0.5).

Decoder: Figure 2 shows the decoder implementation which mirrors Listing 2. Two 16b registers HI and LO maintain the current range whereas a 16*b* CODE register and an 8*b* OFS register are used to read, respectively, from the symbol and the offset streams as values are decoded. The decoder produces a single value per cycle. The current 16b input in CODE is compared against the scaled ranges to find where it lies and which symbol it corresponds to. The ranges are scaled by multiplying the probability count register values (hiCnt[i] with the current range defined by HI and LO. Using a process similar to that of the Encoder, an appropriate number of bits are discarded from CODE and HI and LO are adjusted. The "SYMBOL GEN" block produces the output value by adding enough offset bits to the value read from the base[i] value

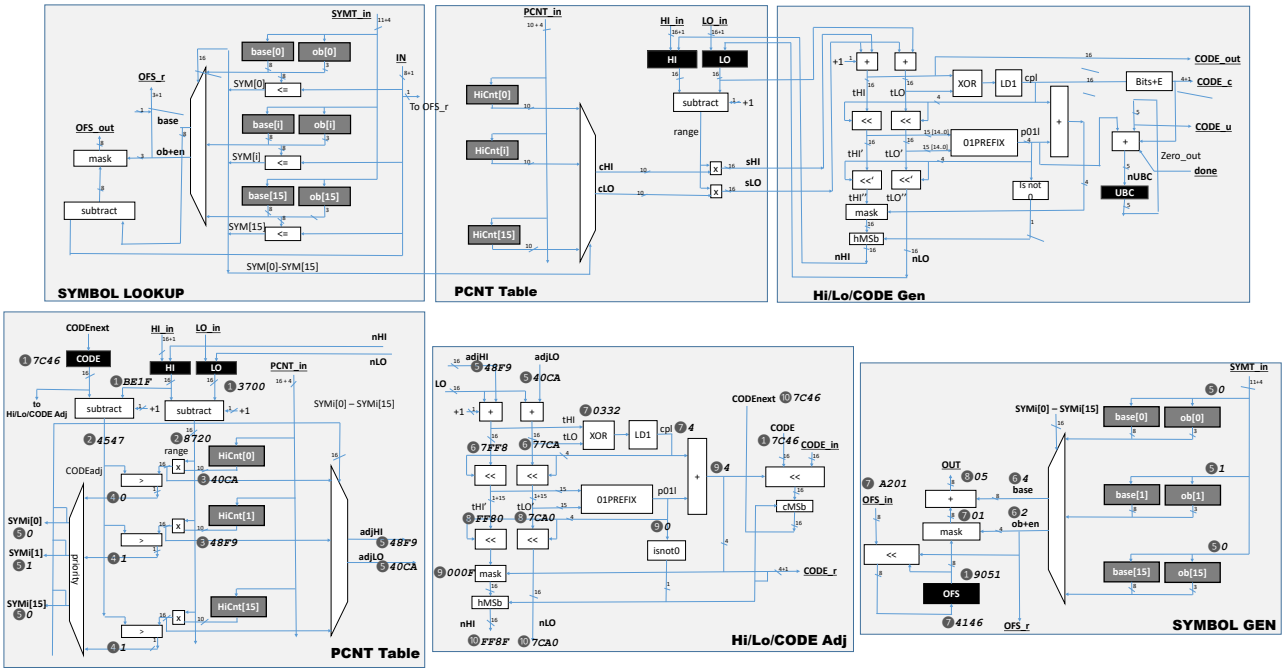

Figure 2. Atalanta Encoder (top) and Decoder (bottom): Filled blocks are registers. Everything else is combinatorial logic.

```
1   HIGH, LOW: 16b regs, initially 0xffff, 0x0000
2   PCNT[]: Symbol & Prob. Count Table (Table 1)
3   CODE: 16b Encoded Symbol Stream Input Register
4   OFS: 16b Offset Stream Input Register
5   OUT: 16b Decoded Value Output
6    Decode_Value:
7     range = HIGH - LOW + 1
8    Table Lookup:
9     find i s.t. range_min≤ val ≤range_max
10    where range_min = (PCNT[i].tlow * range) ≫ 10
11          range_max = (PCNT[i].thigh * range) ≫ 10
12      OL = PCNT[i].OL
13      OUT = PCNT[i].v_min
14   Adjust high & low:
15    HIGH = LOW + range_min - 1
16    LOW = LOW + range_max
17   Consume from Arithmetic Coding Stream:
18    HIGH & LOW common prefix?
19    Shift out from HIGH and LOW
20    vacated LSbs: fill w/ 1s/0s for HIGH/LOW
21    Shift out this many bits from CODE filling in
         from encoded stream
22   Underflow Handling:
23    (HIGH,LOW) of the form x0...0xxx,x1...1xxx)?
24    remove 0...0 and 1...0 from (HIGH,LOW)
25    shift out corresponding bits from CODE
26    vacated LSbs: fill w/ 1s/0s for HIGH/LOW
27    fill in CODE from encoded stream
28    invert MSb of CODE
29   Input from Offset stream:
30    read OL bits from Offset Stream
31    ADD those into OUT
```

Listing 2. Atalanta decoding a value

for the matching range. The figure also shows an example of decoding a value assuming the probability table of Table 1

Table 3. Atalanta Encoder/Decoder: Bits and Functional Units.

	Enc.	Dec.
Register Costs (bits)		
Bits per Element	***Count***	
Probability Table		
HiCnt	***ob***	***base***		
10b	3b	8b	16x	16x
HI / LO		
16b			2x	2x
UBC		
5b			1x	
CODE/OFS		
16b/8b				1x
Totals		
Bits			**373b**	**392b**
Bytes			**47B**	**49B**

	Encoder	Decoder
Functional Units		
Multipliers		**Count**
16b x 10b	2x	16x
Subtractors Adders		
16b	3x	20x
4b	2x	
5b	1x	
8b	1x	1x
Unidirectional Shifters		
16b	4x	4x
8b	1x	1x

4.2 Register Bit and Functional Unit Costs

Table 3 reports the number of register bits and the major combinatorial blocks used by the encoder and decoder. The encoder uses 373 bits (slightly less than 47 bytes) most of which are for the encoding table. The corresponding register bit costs for the decoder are nearly identical. Overall either unit uses few functional units with the encoder using more compared to the decoder. Section 5.2 reports the area and energy of synthesized layouts of the units.

4.3 Encoding/Decoding Table Optimizer

A software optimizer given a tensor generates an optimized probability table, e.g., Table 1). The optimizer is invoked once per tensor. Since weights do not change, a single pass is sufficient. For activations, profiling proved effective since their distribution does not change much. The optimizer has to partition the input value range into sub-ranges so that the

encoded stream is as small as possible. This is a non-trivial balancing act: the wider a sub-range, the less bit space its symbols will use and the more bits its offsets will need.

Without loss of generality this description assumes that the inputs are 8b values. Given a tensor, the process first creates a per value i frequency histogram with 2^8 buckets (not shown in the listing) which becomes the input to `findPT()`. The first step is to initialize the probability table (PT) by uniformly distributing the value range $[0, 2^8 - 1]$ over the entries (line 36 in Listing 3). Function `search()` searches through candidate configurations and returns the best PT it found and its corresponding footprint in bits. Line 40 decides whether the optimizer will try to search for an even better configuration; it does if the last attempt found a PT that reduced size by more than 1%. Search uses recursion whose depth is not allowed to exceed `DEPTH_MAX` (2 was sufficient). The parameter `around` identifies the PT entries `search()` would try to adjust (< 0 means all, otherwise its entries within a distance of 1 from index `around`). Call to By calculating the entropy per range `encoded_size()` estimates the compression ratio possible given the current table configuration.

Generating the Probability Counts: After the v_{min} values are decided, the probability counts are generated. The probability counts range of $[0...2^m]$, where m a design parameter (we use $m = 10$), is partitioned proportional to the frequency of the values in each range given the v_{min} configuration.

Final Adjustment for Activations: For activation tensors, it is possible that some values that do not appear in the profiled inputs may appear for some other inputs. We handle these cases via a post processing step which adjusts the probability count table by "stealing" a single count from another non-zero entry for each zero entry.

```
1  PT : Prob. table, array [1...N]
2  histogram : how many times each input value appears
       , array [0..VALUE_MAX]
3  depth : integer,  around: integer 1...N,
4  DEPTH_MAX : integer, default 2
5  THRESHOLD: real, default 0.99
6  search (histogram, PT, minsize, depth, around)
7   tryPT = PT
8   for i=1 to N
9     if around>=1 and |i - around|!=1: continue
10    save=tryPT[i].v_min
11    repeat
12        if i==1: pv_min=0
13        else:    pv_min=tryPT[i-1].v_min
14        if tryPT[i].v_min==pv_min: break
15        tryPT[i].v_min--
16        if depth > DEPTH_MAX:
17           PT, minsize = search(histogram, tryPT,
       minsize, depth + 1, i)
18        else:
19           trysize = encoded_size(histogram, tryPT)
20           if try_size<minsize: PT=tryPT, minsize=
       trysize
21    tryPT[i].v_min=save
22    repeat
23        if i==N: nv_min=VALUE_MAX
24        else:    nv_min=tryPT[i+1].v_min
25        if tryPT[i].v_min==pv_min: break
26        tryPT[i].v_min++
27        if depth > DEPTH_MAX:
28           PT, best_size = search(histogram, tryPT,
       minsize, depth + 1, i)
29        else:
30           try_size = encoded_size(histogram, tryPT)
31           if try_size<minsize: PT=tryPT, minsize=
       trysize
32    return PT, minsize
33
34 findPT(histogram)
35  initialize PT to uniform distribution
36  repeat
37    size=encoded_size(histogram, PT)
38    PT, newsize = search(histogram, PT, size, 1, -1)
39    if newsize/size>=THRESHOLD: break
40  return PT
```

Listing 3. Probability Table Generation

5 Evaluation

Comparison Points: We compare Atalanta against: 1) Baseline: that does not apply any data compression. 2) Run-Length Encoding (RLE), Run-Length Encoding for Zeros (RLEZ) [14, 29, 54]: run length encoding techniques that encode values as tuples of *(value, distance)* where *distance* is the number of similar values, or zeros, until the next non-similar, or non-zero value for RLE and RLEZ, respectively. The distance is limited to be up to 15 for a maximum overhead of 4-bit per tuple. 3) ShapeShifter [48] which groups the values in a predetermined group size G and dynamically detects the minimal precision P needed to represent the values in the group based on the actual range of values in the group. Then, the group is represented with $(G \times P + \log_2 P_{max})$ bits where P_{max} is the max precision supported per value. We evaluate a variant of ShapeShifter that is optimized for 8-bit models. Section 5.8 compares *Atalanta* with implementations of: 4) Deflate [57], 5) LZMA [101], and Bit Plane Compression [18, 44].

DNN models: We evaluate a set of (quantized to int8 unless otherwise noted) models spanning a wide range of applications (Table 4). The models were obtained directly from the respective sources and are used *unmodified*. From the Torchvision repository, we use those models that have been pre-quantized to int8 [3]. From the IntelAI repository, we similarly use only those models whose weights are quantized [38]. ResNet18-PACT was quantized to 4-bit except for the first and last layers which remain in 8-bit using the PACT method of Choi et al. [16]. ResNet18-Q is pruned with the modified training method of BitPruning to arrive at per layer fixed-point precisions that never exceed 8-bit [67]. The "per-layer" quantized models were quantized further with the profiling-based quantization method of Nikolić et al. [68] which further trims precisions per layer. Finally, we study pruned versions of Alexnet, Googlenet, and Yolo V8.

Trace Collection: We run inference with the original models on an NVIDIA RTX 3090 GPU or an Intel processor as

Table 4. Neural network models studied. PL=Per-Layer, IntelLabs=IntelLabs Distiller. *Atalanta* vs. Shapeshifter

Network	Dataset	Application	Data Type	Quantizer	Table Generation HH:MM:SS	*Atalanta* over Shapeshifter Compression Wgts	Compression Acts	Memory Energy Wgts	Memory Energy Acts
GoogLeNet [87]	ImageNet [80]	Classification	int8	Torchvision	00:22:26	6.8%	24.7%	2.7%	21.4%
Inception v3 [88]	ImageNet	Classification	int8	Torchvision	00:47:31	7.1%	26.7%	3.1%	23.5%
Mobilenet v2 [82]	ImageNet	Classification	int8	Torchvision	00:22:33	8.6%	24.9%	4.7%	21.6%
Mobilenet v3 [35]	ImageNet	Classification	int8	Torchvision	00:34:44	10.1%	29.1%	6.2%	26.0%
Resnet18 [32]	ImageNet	Classification	int8	Torchvision	00:08:25	5.9%	27.8%	1.8%	24.6%
Resnet50 [32]	ImageNet	Classification	int8	Torchvision	00:23:32	7.2%	37.7%	3.2%	35.0%
Resnext101 [96]	ImageNet	Classification	int8	Torchvision	00:53:44	8.1%	37.2%	4.1%	34.4%
Shufflenet v2 [59]	ImageNet	Classification	int8	Torchvision	00:22:26	14.7%	26.1%	4.6%	22.9%
Inception v4 [86]	ImageNet	Classification	int8	IntelAI	00:38:21	10.3%	27.1%	6.4%	24.0%
Mobilenet v1 [36]	ImageNet	Classification	int8	IntelAI	00:04:46	9.7%	N/A	5.7%	N/A
Resnet101 [32]	ImageNet	Classification	int8	IntelAI	00:34:46	11.8%	N/A	8.0%	N/A
R-FCN Resnet101 [35]	COCO [52]	Object Detection	int8	IntelAI	00:35:36	11.7%	N/A	7.9%	N/A
SSD-Resnet34 [58]	COCO	Object Detection	int8	IntelAI	00:34:38	8.8%	N/A	4.9%	N/A
Wide Deep [15]	Kaggle Disp. Adv. [1]	Recommendation	int8	IntelAI	00:01:07	9.1%	N/A	5.1%	N/A
Q8BERT [4, 23]	MRPC [91]	NLP	int8	IntelLabs	00:25:36	12.5%	24.6%	8.7%	21.4%
NCF [33, 102]	ml-20m [31]	Recommendation	int8	IntelLabs+PL	00:05:22	7.8%	39.6%	3.7%	37.0%
ResNet18-PACT [16]	ImageNet	Classification	int4/int8	IntelLabs+PL	00:34:00	49.1%	27.9%	46.9%	24.7%
SSD-MobileNet [55, 76]	COCO	Object Detection	int8	MLPerf+PL	00:21:41	44.8%	23.6%	42.4%	20.2%
MobileNet [36, 76]	ImageNet	Classification	int8	MLPerf	00:11:31	35.2%	18.5%	32.4%	15.0%
bilstm [92]	Flickr8k [75]	Captioning	int8	per-layer	00:10:27	13.1%	14.8%	9.3%	11.1%
SegNet [9]	CamVid [11]	Segmentation	int8	per layer	00:08:31	20.7%	21.7%	17.2%	18.3%
ResNet18-Q [32, 67]	ImageNet	Classification	int8	per-layer	00:26:21	13.2%	30.8%	9.4%	27.7%
AlexNet-Eyeriss [97]	ImageNet	Classification	int8/Pruned	per-layer	00:07:36	67.6%	16.8%	66.1%	13.1%
GoogLeNet-Eyeriss [97]	ImageNet	Classification	int8/Pruned	per layer	00:50:02	44.2%	30.5%	41.7%	27.4%
					Mean	20.5%	27.1%	16.8%	24.0%
BERT-wnli [23]	GLUE WNLI [91]	NLP	int8	Linear	00:00:28 (*)				
EfficientNet V2 [89]	ImageNet	Classification	int8	Neuralmagic [5]	00:00:43 (*)				
GPT2 [73]	Wikitext2[61]	NLP	int8	per-layer	00:00:37 (*)				
Llama2 7B [6, 90]	Meta Research[90]	NLP	int8	GPTQ [26]	00:01:02 (*)				
Yolo V8 [7, 41]	COCO	Object Detection	int8/pruned	Neuralmagic	00:00:42 (*)				

required. PyTorch and TensorFlow layer hooks dump layer input weights and activations into numpy files. Up to 9 input activation samples per layer are used to generate the probability tables for activations. The IntelAI models, as provided, use floating-point activations. Hence, we limit attention only to weights. For models obtained from the SparseZoo [5, 7] we load and execute the ONNX models in PyTorch.

Area and Energy Modeling: We attach 64 *Atalanta* encoders and decoders to a dual-channel 8GB DDR4-3200 memory interface. The energy consumption of off-die memory accesses was modeled using Micron's DRAM power model [62]. For *Atalanta*, the data was compressed using the profiling-based probability tables, then passed through the same DRAM power model while taking the overhead power consumption of *Atalanta* engines into account. To model the area and power consumption of *Atalanta*, the encoder/decoders were implemented in Verilog, synthesized via the Synopsys Design Compiler and layout was produced via Cadence Innovus and for a 65nm TSMC technology which is the best that is available to us due to licensing restrictions. The power consumption overhead of the encoding/decoding engines was estimated by capturing circuit activity using Mentor Graphics' ModelSim which was then passed on to Innovus for post-layout simulation. We model a ShapeShifter configuration optimized for 8-bit models with a single Level-1 unit along with eight Level-2 units per off-die DRAM channel and its encoders/decoders using the same technology node and design flow [48].

5.1 Off-Die Memory Traffic

This section reports the change in off-die memory traffic relative to the baseline (no compression) for *Atalanta*, Shapeshifter [48], RLE, RLEZ [14, 29, 54] and Arithmetic Coding (AC) (no offsets, 256-entry encoding/decoding tables). Figure 3 reports the relative reduction in off-die traffic for activations and weights, respectively. *Atalanta* is robust. It always significantly reduces traffic and outperforms the other methods. The traffic reduction is higher for activations than for weights, except for when the models are pruned.

Torchvision Models: For weights, *Atalanta* reduces traffic to as much 0.65 of the baseline for MobileNet_v3 and to as little as 0.88 for ShuffleNet_v2. Much higher reductions are observed with *Atalanta* for activations: as much as 0.41 of the baseline for ResNext101 and as "little" as 0.55 for MobileNet_v3. There are two reasons for the higher compression of activations: High sparsity and a more skewed distribution.

While most weight values do cluster near zero or near the maximum, many of the intermediate values are also present. When compared to the weight distribution of models that were quantized with different methods, this suggests that the lower bits tend to be noisy, a tell-tale sign that the quantization method used by TorchVision uses the full value range regardless of whether it is needed. Shapeshifter and the run-length-based methods have a harder time coping with this noisy distribution. *Atalanta* outperforms Shapeshifter and the other methods for both weights and even more for activations. Shapeshifter groups values (we used a group of 8

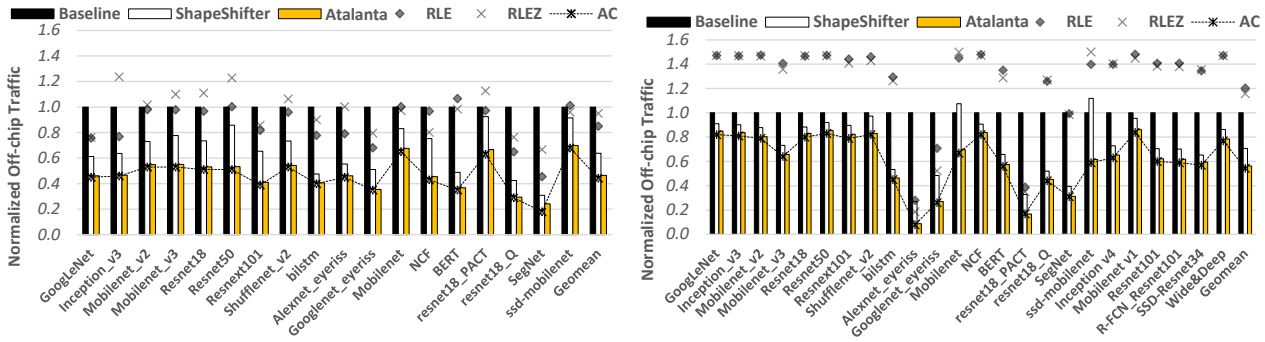

Figure 3. Normalized off-die traffic for the activations (left) and weights (right).

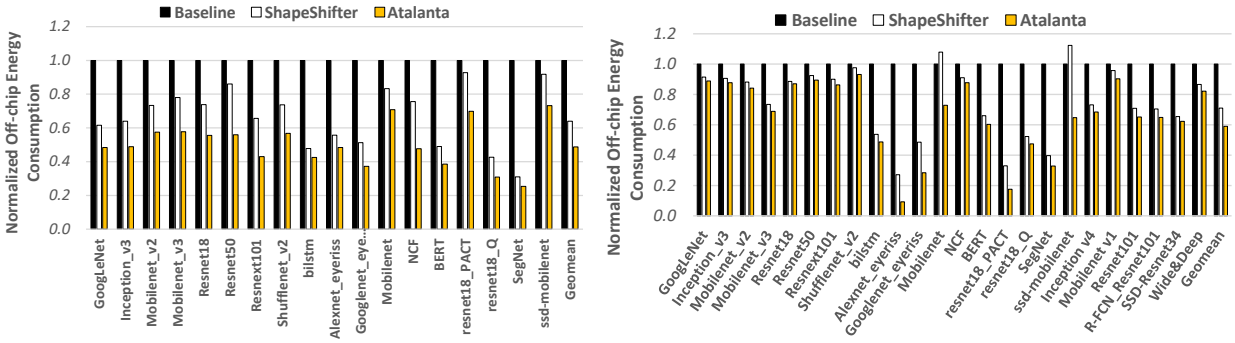

Figure 4. Normalized off-die energy consumption for activations (left) and weights (right).

values as in the original study which we verified that works best for these models) and uses a container that is as larger as necessary to accommodate the largest magnitude value within the group. Encoding efficiency is lost for all other values having a lower magnitude. Instead, *Atalanta* treats each value independently and effectively uses as many bits, even a fractional number, to encode it.

IntelAI Models: These models demonstrate that having a noisy weight distribution is not inherently necessary. Compared to TorchVision, IntelAI's quantization method yields more skewed distributions, a property that *Atalanta* rewards resulting in pronounced compression benefits compared to TorchVision. *Atalanta* reduces traffic to as little as 0.59 of the baseline for SSD-Resnet34 and in the worst case to 0.86 for MobileNet_v1 with most of the models observing a reduction to 0.6. Compared to ShapeShifter, *Atalanta* reduces traffic by 11% on average. *Atalanta* produces considerable benefits for *all* models, even when Shapeshifter fails to do so (e.g., ssd-mobilenet and resnet18_PACT). The run-length-based methods still fail to improve.

Remaining Models: For the remaining models and relative to ShapeShifter, Atalanta compresses activations and weights by 1.34× and 1.51×, respectively. The reduction is higher for weights thanks to their sparse and skewed distribution. *Atalanta* benefits all quantization methods, e.g., it reduces traffic even for the 4b ResNet18_PACT, and much more than ShapeShifter. The run-length-based methods perform best for the pruned models. *Atalanta* is nearly twice as effective.

5.2 Area, Power, and Energy

We model the power consumption of off-die memory via Micron's DRAM power model [62] and use 64 *Atalanta* encoders/decoders. Layout-based measurements are that the 64 *Atalanta* encoders/decoders need a total area of 1.14 mm^2 and consume a total power of 179.2 mW when operating at 680MHz. This power constitutes a 4.7% overhead vs. the power consumed by a dual-channel DDR4-3200 memory system when operating at 90% of its peak bandwidth.

Figure 4 displays energy savings achieved by various compression schemes, normalized to the baseline with no compression. These measurements assume that weights and input activations per layer are read only *once* from off-die memory [85]. Energy savings vary across models in proportion to the compression ratio, with activation distributions playing a significant role. Despite ShapeShifter's lower hardware and power costs, *Atalanta* exhibits superior energy

efficiency due to its higher compression and the significant cost of off-die accesses compared to encoders/decoders.

5.3 Performance and Energy Efficiency in Tensorcore

We assess the end-to-end energy efficiency and speedup of integrating Atalanta with a Tensorcore-based accelerator, configured as specified in Table 5. We compare Atalanta and ShapeShifter for off-die compression to a baseline accelerator that doesn't employ any off-die compression technique. Our evaluation focuses on model traces compatible with the Shapeshifter simulator. While *Atalanta* achieves higher compression ratios compared to ShapeShifter, the advantage of *Atalanta* is less pronounced for these models compared to the remaining models. Therefore, the performance and energy advantages of *Atalanta* over ShapeShifter would have been more significant if we could evaluate all models.

Table 5. Tensorcore-based accelerator configuration.

TensorCore-based Accelerator			
# of TCs	64	Activations Buff.	256KB×16 Banks
TC core	4 × 4 PEs	Weights Buff	256KB×16 Banks
PE MACs/Cycle	4	Output Buff.	256KB×16 Banks
Tech Node	65nm	Frequency	1 GHz
off-die Memory		8GB 2-channel DDR4-3200	
Peak TOPS		8.2 TOPS (int8)	

Figure 5 presents the speedup achieved (represented by bars) when Atalanta enhances the baseline Tensorcore-based accelerator, and it also compares this speedup to that achieved when ShapeShifter enhances the same baseline accelerator. On average, Atalanta accelerates execution by a factor of 1.44×, while ShapeShifter achieves a 1.3× speedup over the baseline. Both methods effectively eliminate stalls during off-die transfers, leading to improved utilization of the accelerator's compute units, particularly for models with low compute per byte ratios that tend to be memory-bound. Across all models, *Atalanta* consistently outperforms ShapeShifter. However, for models like BERT, pruned Alexnet, and GoogleNet, which become completely compute-bound, the execution time advantage of *Atalanta* is minimal. Nevertheless, when considering overall energy efficiency, *Atalanta* surpasses ShapeShifter for all models due to its higher reduction in off-die traffic.

Figure 5 shows (markers) the energy efficiency when Atalanta enhances the baseline Tensorcore-based accelerator and compares it with that of when ShapeShifter enhances the baseline Tensorcore-based accelerator *Atalanta* boosts energy efficiency for all models. Improvements vary per model according to the relative importance of the off-die transfer vs. the on-chip compute energy costs. The higher the fraction due to the former the higher the potential and the benefits from *Atalanta*. On average, *Atalanta* boosts energy efficiency by 1.37× outperforming the 1.23× improvement of ShapeShifter. Not only *Atalanta* improves upon ShapeShifter

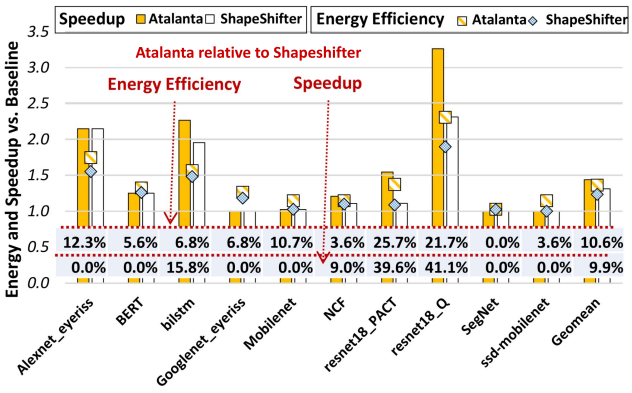

Figure 5. Overall speedup and energy efficiency.

across the board, but more importantly, *Atalanta* delivers benefits even for models that benefit little from ShapeShifter such as ssd-mobilenet, and models with "extreme" quantization such as resnet18_PACT.

5.4 Encoding Table Generation Time

Table 4 reports the total time in HH:MM:SS taken per model to generate the per tensor *Atalanta* encoding tables. Times were measured on a i9-11900K, 64GB DDR4-3200, Ubuntu 20.04.1, Python v3.8.10 system while processing all tensors sequentially. No attempt was made to optimize the *proof-of-concept* implementation of Listing 3. By comparison a C implementation of the algorithm when used for the models marked with (*) took at most 62 seconds to derive the probability tables for Llama2 7B.

5.5 Memory Bandwidth Analysis

Atalanta significantly reduces memory pressure and allows systems to saturate compute units with less capable memory nodes. Figure 6 shows speedup with *Atalanta* compared to without it, using memory speeds from DDR3-1600 to DDR5-7200 and various models. A speedup of 1.0 indicates sufficient memory bandwidth for compute-bound models like resnet18_PACT. For these models, *Atalanta* improves energy efficiency only. For others like Mobilenet, *Atalanta* reduces bandwidth demands and enables maximum compute resource utilization even with slower memories.

5.6 Encoding Table Entries

Figure 7 presents the relative memory footprint with *Atalanta* as the number of encoding table entries varies from 16 (preferred configuration) to 256 (one entry per possible value as in pure AC). We focus on the IntelAI models for brevity. Using 16 entries has a negligible impact on compression effectiveness. As shown in Section 5.2, with 16 entries, 64 *Atalanta* encoders/decoders introduce a power overhead of just 4.7% compared to DRAM. In contrast, the 32, 64, 128, and 256 configurations result in power overheads of 8.6%, 16.3%, 45.1%, and 93.6%, respectively. These findings highlight that

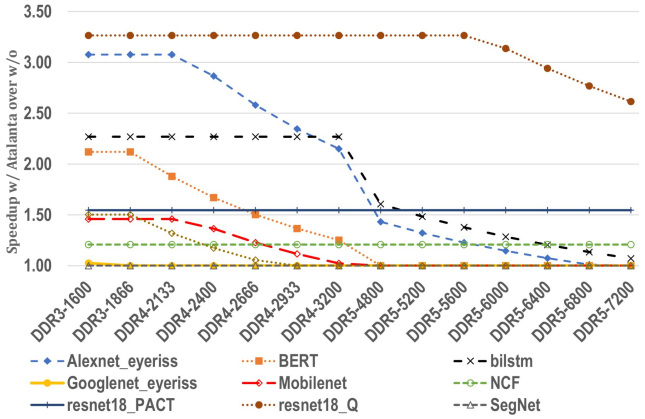

Figure 6. Speedup with *Atalanta* vs. without it for memory speeds ranging from DDR3-1600 up to DDR5-7200 using various models.

pure AC is excessively costly and unsuitable for our needs. Instead, our preferred configuration offers nearly identical compression, while keeping power and area overheads low.

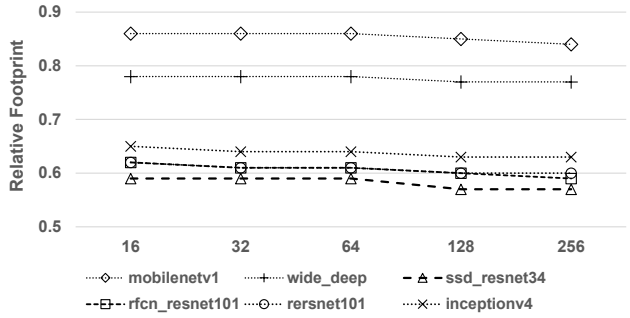

Figure 7. Relative memory footprint reduction with *Atalanta*, when varying the number of encoding table entries.

5.7 Atalanta and Language Models

Figure 8 reports the reduction in footprint with Atalanta on recent transformer-based language models all quantized to int8 as detailed in Table 4. The resulting relative footprints range from 59% to 65%. The highest compression *Atalanta* achieves for the activations of Llama2-7B whose footprint is reduced to 59%. The lowest compression is observed for the activations of GPT2 whose footprint is reduced to 65%.

5.8 Atalanta and Compression

This section compares Atalanta with the following widely used or purpose-built for DL model compression methods: Variants of *Deflate* [22] and *LZMA* [2], and *Bit Plane Compression* (BPC) [18, 44]. We first overview the methods.

Deflate and LZMA: Deflate is a highly-effective compression algorithm widely used where reducing storage footprint overwhelms other considerations (e.g., in data storage). Deflate combines LZ77 compression [101] with, preferably dynamic, Huffman Coding [64]. Informally, in the LZ77 step, a sliding window (typically holding 32KB) stores incoming values and previously seen sequences. When the next sequence matches one in the dictionary, LZ77 encodes it with the dictionary position and length. Hashing is often used to expedite searching [81], but it remains an iterative process. Dynamic Huffman coding improves compression effectiveness by tracking and updating the frequency and mapping of symbols on-the-fly. Deflate implementations differ by how large a sliding window they use, how aggressively they search for pattern matches, and how or if they update the Huffman coding. Lempel-Ziv-Markov (LZMA) is a more aggressive compressor where the dictionary is dynamically sized (up to 4GB) and where pattern searching works at the bit-level [2, 94]. It is considerably more expensive to implement than Deflate.

FPGA-based and ASIC hardware implementations of Deflate that maximize both compression ratio and throughput incur considerable resource overheads, e.g., [12, 28, 49, 50, 83]. Indicatively, a recently reported FPGA implementation requires 69114 LookUp Tables (LUTs), 49779 Flip Flops (FFs) and 521 Block RAMs (BRAMs) of 18Kb each (which are used for the dictionary table in LZ77 compression) [49]. The goal of this work is to reduce the off-die access costs in modern commodity systems while achieving high bandwidth and improving overall energy efficiency.The target solution in our context needs to have low area and energy costs. The high resource and operation costs of hardware-based Deflate implementations do not constitute a good fit for the problem that we tackle. As we demonstrate experimentally below, *Atalanta* rivals and often exceeds the compression possible even with aggressive Deflate configurations.

Bit Plane Compression (BPC): is representative of recently proposed graphics processor unit (GPU) compression methods. It targets a broader class of applications including DL models. It is the underlying compression method used in Buddy Compression [18], which is a technique that amplifies the capacity of GPU memory in two ways: 1) by compressing the data thus requiring less space, and 2) by allowing data to overflow seamlessly from/to CPU memory.

Configurations: Figure 8 reports the relative reduction in off-die traffic by the following methods, normalized to the baseline without any compression: 1) Atalanta, 2) two variants of Deflate [57], $w9l1$ and $w1l8$, 3) two configurations of LZMA, LZMA-l0 and LZMA-l6 corresponding to the least aggressive and the default presets respectively [2], and 4) BPC. We include a few representative Torchvision models, the more recent EfficientNet V2 and Yolo V8 vision models, and the BERT, GPT2 and Llama2 7B transformer-based language models. The buffer costs of Deflate are $2^{winSize+2}+2^{MemLevel+9}$ bytes. We configure $w9l1$ and $w9l8$ with $WinSize = 9$ and 1

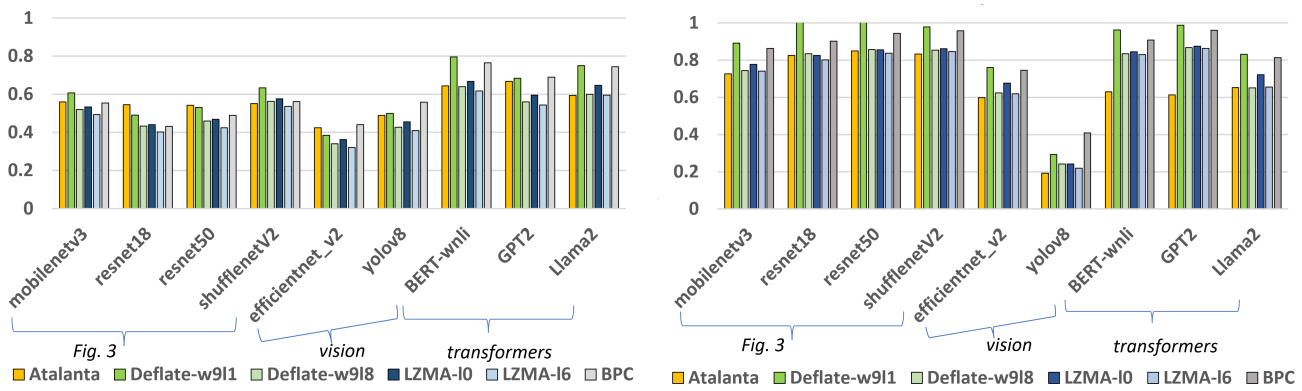

Figure 8. Normalized off-die traffic for activations (left) and weights (right).

and 8 *MemLevel* resulting in memory requirements of $3KB$ and $129KB$, respectively. Configuration $w9l1$ is the least expensive supported by the deflate implementation. We found that increasing *WinSize* or *MemLevel* further did not improve compression by more than 2% compared to $w9l8$. Both configurations use dynamic Huffman coding.

Weights: *Atalanta* rivals or outperforms all other methods except for LZMA-l6 which outperforms *Atalanta* only for resnet18 and resnet50 reducing footprint by an additional 2.4% and 1.2% respectively. *Atalanta* edges out even LZMA-l6 for several models, with the highest differences observed in BERT-wnli, and GPT2 respectively at 20% and 25%. Except for Yolo V8, BPC falls between $w9l1$ and $w1l8$ often being closer to $w9l1$. For Yolo v8, it under-performs yielding almost twice the footprint compared to $w9l8$. *Atalanta* consistently outperforms BPC with the differences being noticeably higher for the transformers and Yolo V8.

Activations: *Atalanta* rivals or outperforms the less aggressive configurations of Deflate and LZMA, while it matches the most aggressive LZMA-l6 for Llamma2. *Atalanta* matches or outperforms $w9l1$ except for the resnets and Efficientnet V2 with the highest difference observed for BERT-wnli and Llama2 at 15.2% and 15.6% respectively. BPC matches or outperforms Atalanta for the earlier vision models (Mobilenet v3 and the resnets) but, for the more recent models, Atalanta reduces footprint more with the difference being as high as 12.0% for BERT-wnli and 15.0% for Llama2.

The results demonstrate that *Atalanta* is robust, reducing memory footprint and traffic consistently, and rivaling or outperforming the much more expensive Deflate and LZMA methods. It is more effective than BPC for weights and for the activations of recent CNNs and transformer models.

5.9 Bit Breakdown and Subrange Optimization

Most bits in Atalanta-encoded tensors are allocated for offsets, as shown in the breakdown in Figure 9 (left y-axis/stacked bars). The figure also illustrates (right y-axis/× marks) the relative improvement in footprint reduction

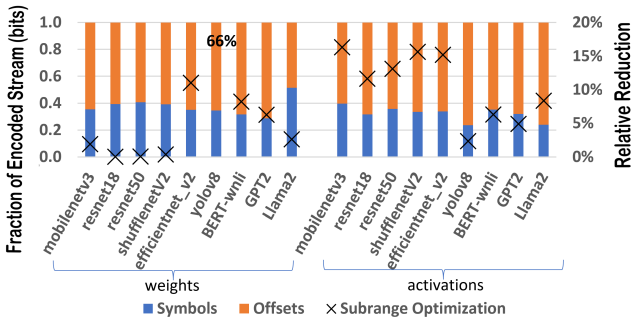

Figure 9. Allocation of Bits in *Atalanta*-encoded streams and improvement with subrange optimization.

achieved by the sub-range selection method of Listing 3 (S_O) compared to uniform sub-range selection (S_U), where, for example, 16 consecutive sub-ranges of 16 values each are used for int8. The metric is defined as $(S_U - S_O)/S_U$, where S_x represents the relative footprint with each method compared to the uncompressed tensor. All models benefit from sub-range optimization with Yolo V8 benefiting the most with its weight footprint reduced from 0.52x with uniform sub-ranges to just 0.19x. Future work may explore eliminating some of the offset bits as long as some loss in value fidelity is tolerable.

5.10 Atalanta During Training

Atalanta can reduce data footprints when *training* for fixed-point inference. Throughout training, weights and activations are kept in fixed-point (e.g., INT8) with only weight updates accumulated in FP. The forward pass stashes activations from *all* layers and across all samples in a batch and backpropagation uses them. Typically, batches comprise 64-256 input samples. We traced the INT8 tensors generated when ResNet18 was trained on ImageNet1K with BitPruning [67]. Training commences with no compression. We generate probability tables at epochs 0, 35, and 65, and after at least 1000 iterations (batch) in the epoch (each epoch has

5005 iterations) using just a single batch. We measure compression for iterations at epochs 1(0), 35(0), 36(35), 65(35), and 66(65). To properly account for changes in the distribution, the measurements are taken at the epochs listed in parentheses, which are far after the table generation points and naturally contain a different mix of input samples. The total footprint per batch is 5GB and *would not fit on-chip*. AC implementation of Listing 3 took at most 21.48s to generate *all* tables at each point. Figure 10 shows the relative footprint reduction achieved with *Atalanta* in total and for activations only compared to baseline without any data compression, when training ResNet18 model. *Atalanta* significantly reduces the memory footprint. The measurement at epoch 35 (E35) shows that even when using a table generated far in the past at epoch 0, *Atalanta* still reduces footprint to 1/3. Regenerating the tables boosts compression efficiency and the footprint hovers around 40% from then on.

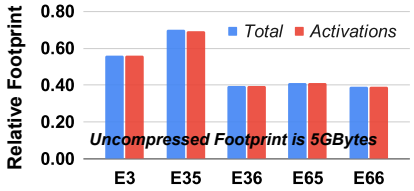

Figure 10. Relative footprint of *Atalanta* when training ResNet18.

5.11 Atalanta and BFloat16

Atalanta can compress the *exponents* during inference with floating-point, since they exhibit very biased distributions. We measure compression for BERT Base [23] (MNLI task [95], GLUE dataset [91], 128 token sequence) and Roberta Large [56] (Question Answering task, SQuADv1 dataset [74], 384 token sequence), when using BFloat16 weights and activations. Table 6 reports that *Atalanta* reduces the exponent footprint to nearly 1/3 of the original. Exponents in BFloat16 represent 50% of the total footprint. Training with a technique such as Quantum Mantissa can trim the mantissas [69].

Table 6. *Atalanta* Footprints for BFloat16 Exponents

Model	Activations	Weights	Overall
BERT Base	0.32	0.34	0.34
Roberta Large	0.39	0.34	0.36

5.12 Atalanta and Indirect Quantization

After examining models with direct quantization, we now explore *Atalanta*'s interaction with indirect quantization.

Deep Compression (DC) [30] employs 16 centroid values and Huffman coding for dictionary indexes, focusing on weights and 16-bit models. UNIQ [10] learns centroids through network retraining, using signal processing theory to treat quantization effects as noise. Unlike DC, UNIQ's dictionary indexes vary from 4-bit to 32-bit, based on tolerable accuracy loss.

Table 7. Deep Compression Quantized Models: Additional memory Footprint reduction with *Atalanta* over (enhanced) Deep Compression: Whole Network, and Per Layer Maximum and Minimum

	Activations			Weights		
	Total	**Min**	**Max**	**Total**	**Min**	**Max**
resnet50	19.12%	2.11%	68.77%	1.02%	0.12%	1.11%
mobilenet v3	7.35%	0.69%	2.70%	3.30%	0.89%	24.33%

To explore the interplay of *Atalanta* with dictionary methods, we simulate a DC++ method that optimistically assumes: 1) DC can be used with 8b models, 2) DC can be used for activations, and 3) there is no compromise in accuracy. Table 7 demonstrates the additional memory reduction (D) achieved by *Atalanta* on DC++-quantized ResNet50 and MobileNet_v3. D is defined as $D = S_{w/oAtalanta} - S_{w/Atalanta}$, where S_x is relative to the base footprints (e.g., 0.75x). We report the relative memory footprint reduction for the entire network, the maximum reduction at layer granularity, and the minimum reduction at layer granularity. *Atalanta* outperforms DC++ for two reasons: a) *Atalanta* generates 16 ranges, each containing a single centroid without requiring offset bits, and b) *Atalanta* employs arithmetic coding, which is superior to the Huffman coding used by DC when index frequencies are not of the form 2^{-k}. Since the centroids are 8b integers, *Atalanta* can directly compress them, eliminating the need for a dictionary.

GOBO [99] and Mokey [100] are post-training, dictionary-based quantizers for the floating-point tensors of transformers. Unlike DC, both segregate values into outliers and non-outliers. GOBO focuses solely on weights, retaining outliers in floating-point. Mokey,quantizes all values into two 8-entry int16 dictionaries per tensor where the centroids fit a $a^{index} + b$ function. The closed-form relationship among centroids allows Mokey to avoid implementing dictionaries for converting indexes to centroid values. Instead, Mokey can perform all computations directly on the indexes themselves. While both techniques induce some accuracy loss, they outperform many methods that quantize to 8 bits. They both utilize fixed-point indexes to represent values.

We applied GOBO and Mokey to quantize BERT-Base, compressing 3b and 4b indexes with *Atalanta*. Table 8 shows *Atalanta*'s additional footprint reduction D over GOBO and Mokey. The reductions, more significant for activations, demonstrate *Atalanta*'s adaptability in minimizing footprint across different quantization methods.

We conclude that *Atalanta* matched or exceeded benefits when used over the dictionary-quantized models without further effort. With *Atalanta*, designers can selectively deploy quantization methods without being "locked" into them.

Table 8. *Atalanta* over GOBO or Mokey on BERT-Base.

	Weights			Activations		
	min	**max**	**Total**	**min**	**max**	**Total**
GOBO	2.3%	4.9%	3.0%	N/A	N/A	N/A
Mokey	2.3%	5.0%	2.4%	2.1%	80.1%	26.2%

6 Other Related Work

The *compression potential* of entropy encoding for tensor values has been investigated using software-based adaptive AC over *recorded* tensor traces *ex post facto* [93]. Prior works that implement AC either maximize compression at the expense of high cost and complexity [98], or improve implementation efficiency and practicality to some degree [37, 66]. In contrast, *Atalanta* proposes an effective variation of AC that can both be implemented in hardware at low cost (See Section 5.2) and achieve near-optimal compression efficiency (Figure 3).

7 Conclusion

Atalanta is a practical and lossless tensor compression method that provides high system performance and energy-efficiency in deep learning inference and training. It enables transparent and highly-efficient encoding for weights and activations, it is low-cost and can be seamlessly integrated with state-of-the-art deep learning accelerators. It is general to support a wide variety of DL models (both quantized and not) and rewards further quantization and pruning without forcing a particular quantization choice or pruning contraints up-front.

Acknowledgments

This work was supported in part by an NSERC Discovery Grant and the NSERC COHESA Research Network. We thank our shepherd, Moumita Dey, and the anonymous reviewers for their feedback. We are grateful that the ASPLOS review process allowed us to engage in meaningful discussion with the reviewers, to address concerns and offer clarifications, and to improve the descriptions and the evaluation section. Code for Atalanta can be found at `https://github.com/moshovos/Atalanta` and an earlier write-up is available at Arxiv [47]. The University of Toronto maintains all rights on the technologies described herein.

References

[1] "Kaggle display advertising challenge." [Online]. Available: https://www.kaggle.com/c/criteo-display-ad-challenge/data

[2] "LZMA SDK (Software Development Kit) — 7-zip.org," https://7-zip.org/sdk.html, [Accessed 10-08-2023].

[3] "Torchvision." [Online]. Available: https://pytorch.org/vision/stable/index.html

[4] "Nlp architect by intel ai lab," Nov. 2018. [Online]. Available: https://doi.org/10.5281/zenodo.1477518

[5] 2023. [Online]. Available: https://sparsezoo.neuralmagic.com/models/efficientnet_v2-s-imagenet-base_quantized

[6] 2023. [Online]. Available: https://huggingface.co/decapoda-research/llama-7b-hf

[7] 2023. [Online]. Available: https://sparsezoo.neuralmagic.com/models/yolov8-m-coco-pruned75_quantized

[8] N. Abramson, *Information theory and coding*. McGraw-Hill Book Co, 1963.

[9] V. Badrinarayanan, A. Kendall, and R. Cipolla, "SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2017.

[10] C. Baskin, N. Liss, E. Schwartz, E. Zheltonozhskii, R. Giryes, A. M. Bronstein, and A. Mendelson, "UNIQ: uniform noise injection for non-uniform quantization of neural networks," *ACM Trans. Comput. Syst.*, vol. 37, no. 1-4, pp. 4:1–4:15, 2019. [Online]. Available: https://doi.org/10.1145/3444943

[11] G. J. Brostow, J. Fauqueur, and R. Cipolla, "Semantic object classes in video: A high-definition ground truth database," *Pattern Recognition Letters*, vol. 30, no. 2, pp. 88 – 97, 2009, video-based Object and Event Analysis. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167865508001220

[12] CAST Inc., "CAST ZipAccel-C Intel GZIP/ZLIB/Deflate Data Compression Core ," 2022. [Online]. Available: https://www.cast-inc.com/compression/gzip-data-compression/zipaccel-c

[13] L. Cavigelli and L. Benini, "Extended Bit-Plane Compression for Convolutional Neural Network Accelerators," in *Proc. IEEE AICAS*, 2018.

[14] Chen, Yu-Hsin and Krishna, Tushar and Emer, Joel and Sze, Vivienne, "Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks," in *IEEE International Solid-State Circuits Conference, ISSCC 2016, Digest of Technical Papers*, 2016, pp. 262–263.

[15] H.-T. Cheng, L. Koc, J. Harmsen, T. Shaked, T. Chandra, H. Aradhye, G. Anderson, G. Corrado, W. Chai, M. Ispir, R. Anil, Z. Haque, L. Hong, V. Jain, X. Liu, and H. Shah, "Wide & deep learning for recommender systems," 2016.

[16] J. Choi, Z. Wang, S. Venkataramani, P. I. Chuang, V. Srinivasan, and K. Gopalakrishnan, "PACT: parameterized clipping activation for quantized neural networks," *CoRR*, vol. abs/1805.06085, 2018. [Online]. Available: http://arxiv.org/abs/1805.06085

[17] E. Choukse, M. Erez, and A. R. Alameldeen, "Compresso: Pragmatic main memory compression," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 546–558.

[18] E. Choukse, M. B. Sullivan, M. O'Connor, M. Erez, J. Pool, D. Nellans, and S. W. Keckler, "Buddy compression: Enabling larger memory for deep learning and hpc workloads on gpus," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 926–939.

[19] M. Courbariaux, Y. Bengio, and J.-P. David, "BinaryConnect: Training Deep Neural Networks with binary weights during propagations," *ArXiv e-prints*, Nov. 2015.

[20] M. Courbariaux, Y. Bengio, and J.-P. David, "Binaryconnect: Training deep neural networks with binary weights during propagations," in *Advances in Neural Information Processing Systems*, 2015, pp. 3123–3131.

[21] B. Dally, "Power, programmability, and granularity: The challenges of exascale computing," in *2011 IEEE International Parallel Distributed Processing Symposium*, May 2011, pp. 878–878.

[22] L. P. Deutsch, "DEFLATE Compressed Data Format Specification version 1.3," RFC 1951, May 1996. [Online]. Available: https://www.rfc-editor.org/info/rfc1951

[23] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," *CoRR*, vol. abs/1810.04805, 2018. [Online]. Available: http://arxiv.org/abs/1810.04805

[24] L. Durant, O. Giroux, M. Harris, and N. Stam, "Nvidia developer blog," May 2017. [Online]. Available: https://devblogs.nvidia.com/inside-

volta/

[25] I. Edo Vivancos, S. Sharify, D. Ly-Ma, A. Abdelhadi, C. Bannon, M. Nikolic, M. Mahmoud, A. Delmas Lascorz, G. Pekhimenko, A. Moshovos, and et al., "Boveda: Building an on-chip deep learning memory hierarchy brick by brick," Mar 2021. [Online]. Available: https://proceedings.mlsys.org/paper/2021/hash/013d407166ec4fa56eb1e1f8cbe183b9-Abstract.html

[26] E. Frantar, S. Ashkboos, T. Hoefler, and D. Alistarh, "Gptq: Accurate post-training quantization for generative pre-trained transformers," 2023.

[27] A. Gholami, S. Kim, Z. Dong, Z. Yao, M. W. Mahoney, and K. Keutzer, "A survey of quantization methods for efficient neural network inference," *CoRR*, vol. abs/2103.13630, 2021. [Online]. Available: https://arxiv.org/abs/2103.13630

[28] C. Giannoula, K. Huang, J. Tang, N. Koziris, G. Goumas, Z. Chishti, and N. Vijaykumar, "Daemon: Architectural support for efficient data movement in fully disaggregated systems," *Proc. ACM Meas. Anal. Comput. Syst.*, 2023.

[29] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: efficient inference engine on compressed deep neural network," in *43rd ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2016, Seoul, South Korea, June 18-22, 2016*, 2016, pp. 243–254.

[30] S. Han, H. Mao, and W. J. Dally, "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding," *arXiv:1510.00149 [cs]*, Oct. 2015, arXiv: 1510.00149. [Online]. Available: http://arxiv.org/abs/1510.00149

[31] F. M. Harper and J. A. Konstan, "The movielens datasets: History and context," *ACM Trans. Interact. Intell. Syst.*, vol. 5, no. 4, Dec. 2015. [Online]. Available: https://doi.org/10.1145/2827872

[32] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015. [Online]. Available: http://arxiv.org/abs/1512.03385

[33] X. He, L. Liao, H. Zhang, L. Nie, X. Hu, and T.-S. Chua, "Neural collaborative filtering," in *Proceedings of the 26th International Conference on World Wide Web*, ser. WWW '17. Republic and Canton of Geneva, CHE: International World Wide Web Conferences Steering Committee, 2017, p. 173–182. [Online]. Available: https://doi.org/10.1145/3038912.3052569

[34] M. Horowitz, "1.1 computing's energy problem (and what we can do about it)," vol. 57, 02 2014, pp. 10–14.

[35] A. Howard, M. Sandler, G. Chu, L.-C. Chen, B. Chen, M. Tan, W. Wang, Y. Zhu, R. Pang, V. Vasudevan, Q. V. Le, and H. Adam, "Searching for mobilenetv3," 2019.

[36] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *CoRR*, vol. abs/1704.04861, 2017. [Online]. Available: http://arxiv.org/abs/1704.04861

[37] P. G. Howard and J. S. Vitter, "Practical implementations of arithmetic coding," *The Kluwer International Series in Engineering and Computer Science*, p. 85–112, 1992.

[38] IntelAI, "Models/benchmarks at master · intelai/models." [Online]. Available: https://github.com/IntelAI/models/tree/master/benchmarks

[39] J.-W. Jang, S. Lee, D. Kim, H. Park, A. S. Ardestani, Y. Choi, C. Kim, Y. Kim, H. Yu, H. Abdel-Aziz, J.-S. Park, H. Lee, D. Lee, M. W. Kim, H. Jung, H. Nam, D. Lim, S. Lee, J.-H. Song, S. Kwon, J. Hassoun, S. Lim, and C. Choi, "Sparsity-aware and re-configurable npu architecture for samsung flagship mobile soc," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 15–28.

[40] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza, "Dissecting the NVIDIA volta GPU architecture via microbenchmarking," *CoRR*, vol. abs/1804.06826, 2018. [Online]. Available: http://arxiv.org/abs/1804.06826

[41] G. Jocher, A. Chaurasia, and J. Qiu, "Ultralytics yolov8," 2023. [Online]. Available: https://github.com/ultralytics/ultralytics

[42] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. New York, NY, USA: ACM, 2017, pp. 1–12. [Online]. Available: http://doi.acm.org/10.1145/3079856.3080246

[43] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, N. E. Jerger, and A. Moshovos, "Proteus: Exploiting numerical precision variability in deep neural networks," in *Proceedings of the 2016 International Conference on Supercomputing*, ser. ICS '16. New York, NY, USA: ACM, 2016, pp. 23:1–23:12. [Online]. Available: http://doi.acm.org/10.1145/2925426.2926294

[44] J. Kim, M. Sullivan, E. Choukse, and M. Erez, "Bit-plane compression: Transforming data for better compression in many-core architectures," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA '16. IEEE Press, 2016, p. 329–340. [Online]. Available: https://doi.org/10.1109/ISCA.2016.37

[45] S. Kim, A. Gholami, Z. Yao, M. W. Mahoney, and K. Keutzer, "I-bert: Integer-only bert quantization," *International Conference on Machine Learning (Accepted)*, 2021.

[46] G. G. Langdon, "An introduction to arithmetic coding," *IBM Journal of Research and Development*, vol. 28, no. 2, pp. 135–149, 1984.

[47] A. D. Lascorz, M. Mahmoud, and A. Moshovos, "Apack: Off-chip, lossless data compression for efficient deep learning inference," *CoRR*, vol. abs/2201.08830, 2022. [Online]. Available: https://arxiv.org/abs/2201.08830

[48] A. D. Lascorz, S. Sharify, I. Edo, D. M. Stuart, O. M. Awad, P. Judd, M. Mahmoud, M. Nikolic, K. Siu, Z. Poulos, and A. Moshovos, "Shapeshifter: Enabling fine-grain data width adaptation in deep learning," in *Proceedings of the 52Nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: ACM, 2019, pp. 28–41. [Online]. Available: http://doi.acm.org/10.1145/3352460.3358295

[49] M. Ledwon, B. Cockburn, and J. Han, "High-throughput fpga-based hardware accelerators for deflate compression and decompression using high-level synthesis," *IEEE Access*, vol. 8, pp. 62 207–62 217, 03 2020.

[50] M. Ledwon, B. F. Cockburn, and J. Han, "Design and evaluation of an fpga-based hardware accelerator for deflate data decompression," in *2019 IEEE Canadian Conference of Electrical and Computer Engineering (CCECE)*, 2019, pp. 1–6.

[51] T. Liang, J. Glossner, L. Wang, and S. Shi, "Pruning and quantization for deep neural network acceleration: A survey," *CoRR*, vol. abs/2101.09671, 2021. [Online]. Available: https://arxiv.org/abs/2101.09671

[52] T.-Y. Lin, M. Maire, S. Belongie, L. Bourdev, R. Girshick, J. Hays, P. Perona, D. Ramanan, C. L. Zitnick, and P. Dollár, "Microsoft coco: Common objects in context," 2014.

[53] J. Liu, S. Tripathi, U. Kurup, and M. Shah, "Pruning algorithms to accelerate convolutional neural networks for edge applications: A survey," *CoRR*, vol. abs/2005.04275, 2020. [Online]. Available: https://arxiv.org/abs/2005.04275

[54] S. Liu, Z. Du, J. Tao, D. Han, T. Luo, Y. Xie, Y. Chen, and T. Chen, "Cambricon: An instruction set architecture for neural networks," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016, pp. 393–405.

[55] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "Ssd: Single shot multibox detector," *Lecture Notes in Computer Science*, p. 21–37, 2016. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-46448-0_2

[56] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," *arXiv preprint arXiv:1907.11692*, 2019.

[57] J. loup Gailly and M. Adler, "Zlib: A massively spiffy yet delicately unobtrusive compression library," 2023. [Online]. Available: https://www.zlib.net/

[58] X. Lu, X. Kang, S. Nishide, and F. Ren, "Object detection based on ssd-resnet," in *2019 IEEE 6th International Conference on Cloud Computing and Intelligence Systems (CCIS)*, 2019, pp. 89–92.

[59] N. Ma, X. Zhang, H.-T. Zheng, and J. Sun, "Shufflenet v2: Practical guidelines for efficient cnn architecture design," 2018.

[60] M. Mahmoud, K. Siu, and A. Moshovos, "Diffy: A dÉjÀ vu-free differential deep neural network accelerator," in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-51. Piscataway, NJ, USA: IEEE Press, 2018, pp. 134–147. [Online]. Available: https://doi.org/10.1109/MICRO.2018.00020

[61] S. Merity, C. Xiong, J. Bradbury, and R. Socher, "Pointer sentinel mixture models," *CoRR*, vol. abs/1609.07843, 2016. [Online]. Available: http://arxiv.org/abs/1609.07843

[62] I. Micron Technology, "DDR4 power calculator 4.0," https://www.micron.com/~/media/documents/products/power-calculator/ddr4_power_calc.xlsm.

[63] S. Mittal and J. S. Vetter, "A survey of architectural approaches for data compression in cache and main memory systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 5, pp. 1524–1536, 2016.

[64] A. Moffat, "Huffman coding," *ACM Computing Surveys (CSUR)*, vol. 52, no. 4, pp. 1–35, 2019.

[65] M. Nelson, "Arithmetic Coding + Statistical Modeling = Data Compression," *Dr. Dobb's Journal*, vol. February, 1991.

[66] Q.-L. Nguyen, D.-L. Tran, D.-H. Bui, D.-T. Mai, and X.-T. Tran, "Efficient binary arithmetic encoder for hevc with multiple bypass bin processing," in *2017 7th International Conference on Integrated Circuits, Design, and Verification (ICDV)*, 2017, pp. 82–87.

[67] M. Nikolić, G. B. Hacene, C. Bannon, A. D. Lascorz, M. Courbariaux, Y. Bengio, V. Gripon, and A. Moshovos, "Bitpruning: Learning bitlengths for aggressive and accurate quantization," 2020.

[68] M. Nikolić, M. Mahmoud, A. Moshovos, Y. Zhao, and R. Mullins, "Characterizing sources of ineffectual computations in deep learning networks," in *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2019, pp. 165–176.

[69] M. Nikolić, E. T. Sanchez, J. Wang, A. H. Zadeh, M. Mahmoud, A. Abdelhadi, and A. Moshovos, "Schrödinger's fp: Dynamic adaptation of floating-point containers for deep learning training," 2022. [Online]. Available: https://arxiv.org/abs/2204.13666

[70] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "Scnn: An accelerator for compressed-sparse convolutional neural networks," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. New York, NY, USA: ACM, 2017, pp. 27–40. [Online]. Available: http://doi.acm.org/10.1145/3079856.3080254

[71] E. Park, D. Kim, and S. Yoo, "Energy-efficient neural network accelerator based on outlier-aware low-precision computation," in *ISCA*. IEEE Computer Society, 2018, pp. 688–698.

[72] G. G. Pekhimenko, "Practical Data Compression for Modern Memory Hierarchies," 2016.

[73] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," 2019.

[74] P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang, "Squad: 100,000+ questions for machine comprehension of text," in *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, 2016, pp. 2383–2392.

[75] C. Rashtchian, P. Young, M. Hodosh, and J. Hockenmaier, "Collecting image annotations using amazon's mechanical turk," in *Proceedings of the NAACL HLT 2010 Workshop on Creating Speech and Language Data with Amazon's Mechanical Turk*, ser. CSLDAMT '10. Stroudsburg, PA, USA: Association for Computational Linguistics, 2010, pp. 139–147. [Online]. Available: http://dl.acm.org/citation.cfm?id=1866696.1866717

[76] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C.-J. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou, R. Chukka, C. Coleman, S. Davis, P. Deng, G. Diamos, J. Duke, D. Fick, J. S. Gardner, I. Hubara, S. Idgunji, T. B. Jablin, J. Jiao, T. S. John, P. Kanwar, D. Lee, J. Liao, A. Lokhmotov, F. Massa, P. Meng, P. Micikevicius, C. Osborne, G. Pekhimenko, A. T. R. Rajan, D. Sequeira, A. Sirasao, F. Sun, H. Tang, M. Thomson, F. Wei, E. Wu, L. Xu, K. Yamada, B. Yu, G. Yuan, A. Zhong, P. Zhang, and Y. Zhou, "Mlperf inference benchmark," 2019.

[77] M. Rhu, M. O'Connor, N. Chatterjee, J. Pool, Y. Kwon, and S. W. Keckler, "Compressing DMA engine: Leveraging activation sparsity for training deep neural networks," in *IEEE International Symposium on High Performance Computer Architecture, HPCA 2018, Vienna, Austria, February 24-28, 2018*, 2018, pp. 78–91. [Online]. Available: https://doi.org/10.1109/HPCA.2018.00017

[78] J. Rissanen and G. G. Langdon, "Arithmetic coding," *IBM Journal of Research and Development*, vol. 23, no. 2, pp. 149–162, 1979.

[79] J. J. Rissanen, "Generalized kraft inequality and arithmetic coding," *IBM Journal of Research and Development*, vol. 20, no. 3, pp. 198–203, 1976.

[80] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *arXiv:1409.0575 [cs]*, Sep. 2014, arXiv: 1409.0575.

[81] K. Sadakane and H. Imai, "Improving the speed of lz77 compression by hashing and suffix sorting," *IEICE transactions on fundamentals of electronics, communications and computer sciences*, vol. 83, no. 12, pp. 2689–2698, 2000.

[82] M. Sandler, A. G. Howard, M. Zhu, A. Zhmoginov, and L. Chen, "Inverted residuals and linear bottlenecks: Mobile networks for classification, detection and segmentation," *CoRR*, vol. abs/1801.04381, 2018. [Online]. Available: http://arxiv.org/abs/1801.04381

[83] A. Saporito, "The IBM z15 processor chip set," in *IEEE Hot Chips 32 Symposium, HCS 2020, Palo Alto, CA, USA, August 16-18, 2020*. IEEE, 2020, pp. 1–17. [Online]. Available: https://doi.org/10.1109/HCS49909.2020.9220508

[84] S. Sardashti, A. Arelakis, P. Stenstom, and D. A. Wood, *A Primer on Compression in the Memory Hierarchy*, 2015.

[85] K. Siu, D. M. Stuart, M. Mahmoud, and A. Moshovos, "Memory requirements for convolutional neural network hardware accelerators," in *IEEE International Symposium on Workload Characterization*, 2018.

[86] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. Alemi, "Inception-v4, inception-resnet and the impact of residual connections on learning," 2016.

[87] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015, pp. 1–9.

[88] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," 2015.

[89] M. Tan and Q. V. Le, "Efficientnetv2: Smaller models and faster training," *CoRR*, vol. abs/2104.00298, 2021. [Online]. Available: https://arxiv.org/abs/2104.00298

[90] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, D. Bikel, L. Blecher, C. C. Ferrer, M. Chen, G. Cucurull, D. Esiobu, J. Fernandes, J. Fu, W. Fu, B. Fuller, C. Gao, V. Goswami, N. Goyal, A. Hartshorn, S. Hosseini, R. Hou, H. Inan, M. Kardas, V. Kerkez, M. Khabsa, I. Kloumann, A. Korenev, P. S. Koura, M.-A. Lachaux, T. Lavril, J. Lee, D. Liskovich, Y. Lu, Y. Mao, X. Martinet, T. Mihaylov, P. Mishra, I. Molybog, Y. Nie, A. Poulton, J. Reizenstein, R. Rungta, K. Saladi, A. Schelten, R. Silva, E. M. Smith, R. Subramanian, X. E. Tan, B. Tang, R. Taylor, A. Williams, J. X. Kuan, P. Xu, Z. Yan, I. Zarov, Y. Zhang, A. Fan, M. Kambadur, S. Narang, A. Rodriguez, R. Stojnic, S. Edunov, and T. Scialom, "Llama 2: Open foundation and fine-tuned chat models," 2023.

[91] A. Wang, A. Singh, J. Michael, F. Hill, O. Levy, and S. R. Bowman, "GLUE: A multi-task benchmark and analysis platform for natural language understanding," *CoRR*, vol. abs/1804.07461, 2018. [Online]. Available: http://arxiv.org/abs/1804.07461

[92] C. Wang, H. Yang, C. Bartz, and C. Meinel, "Image captioning with deep bidirectional lstms," in *Proceedings of the 2016 ACM on Multimedia Conference*. ACM, 2016, pp. 988–997.

[93] S. Wiedemann, H. Kirchhoffer, S. Matlage, P. Haase, A. Marbán, T. Marinc, D. Neumann, A. Osman, D. Marpe, H. Schwarz, T. Wiegand, and W. Samek, "Deepcabac: Context-adaptive binary arithmetic coding for deep neural network compression," *CoRR*, vol. abs/1905.08318, 2019. [Online]. Available: http://arxiv.org/abs/1905.08318

[94] Wikipedia, "Lempel–Ziv–Markov chain algorithm — Wikipedia, the free encyclopedia," http://en.wikipedia.org/w/index.php?title=Lempel%E2%80%93Ziv%E2%80%93Markov%20chain%20algorithm&oldid=1160826469, 2023, [Online; accessed 18-September-2023].

[95] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, and J. Brew, "Huggingface's transformers: State-of-the-art natural language processing," *ArXiv*, vol. abs/1910.03771, 2019.

[96] S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He, "Aggregated residual transformations for deep neural networks," *arXiv preprint arXiv:1611.05431*, 2016.

[97] Yang, Tien-Ju and Chen, Yu-Hsin and Sze, Vivienne, "Designing Energy-Efficient Convolutional Neural Networks using Energy-Aware Pruning," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[98] Y. Yoo, Y. G. Kwon, and A. Ortega, "Embedded image-domain compression using context models," in *Proceedings 1999 International Conference on Image Processing (Cat. 99CH36348)*, vol. 1, 1999, pp. 477–481 vol.1.

[99] A. H. Zadeh, I. Edo, O. M. Awad, and A. Moshovos, "GOBO: quantizing attention-based NLP models for low latency and energy efficient inference," in *53rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2020, Athens, Greece, October 17-21, 2020*. IEEE, 2020, pp. 811–824. [Online]. Available: https://doi.org/10.1109/MICRO50266.2020.00071

[100] A. H. Zadeh, M. Mahmoud, A. Abdelhadi, and A. Moshovos, "Mokey: enabling narrow fixed-point inference for out-of-the-box floating-point transformer models," in *ISCA '22: The 49th Annual International Symposium on Computer Architecture, New York, New York, USA, June 18 - 22, 2022*, V. Salapura, M. Zahran, F. Chong, and L. Tang, Eds. ACM, 2022, pp. 888–901. [Online]. Available: https://doi.org/10.1145/3470496.3527438

[101] J. Ziv and A. Lempel, "A Universal Algorithm for Sequential Data Compression," *IEEE Transactions on Information Theory*, 1977.

[102] N. Zmora, G. Jacob, L. Zlotnik, B. Elharar, and G. Novik, "Neural network distiller," Jun. 2018. [Online]. Available: https://doi.org/10.5281/zenodo.1297430