Be Like Water: Adaptive Floating Point for Machine Learning

Thomas Y. Yeh [1] Maxwell R. Sterner [1] Zerlina Lai [2] Brandon Y. Chuang [3] Alexander Ihler [4]

Abstract

In the pursuit of optimizing memory and compute density to accelerate machine learning applications, reduced precision training and inference has been an active area of research. While some approaches selectively apply low precision computations, this may require costly off-chip data transfers or mixed precision support. In this paper, we propose a novel numerical representation, Adaptive Floating Point (AFP), that dynamically adjusts to the characteristics of deep learning data. AFP requires no changes to the model topology, requires no additional training, and applies to *all* layers of DNN models. We evaluate AFP on a spectrum of representative models in computer vision and NLP, and show that our technique enables ultra-low precision inference of deep learning models while providing accuracy comparable to full precision inference. By dynamically adjusting to ML data, AFP increases memory density by 1.6x, 1.6x, and 3.2x and compute density by 4x, 1.3x, and 12x when compared to BFP, BFloat16, and FP32.

1. Introduction

In the pursuit of optimizing compute density while reducing memory bandwidth requirements, a number of recent works have examined reduced precision computations for floating point (FP) operations during the training and inference of machine learning models. Two extremes in this spectrum are the fixed-point format (De Sa et al., 2017; Sa et al., 2018) and the 4-bit DNN training format (Sun et al., 2020). In the fixed-point integer format, all exponent bits are removed, resulting in a fixed, implicit exponent. In the case of the 4-bit (DNN) training format, all available bits are used to indicate the sign and the exponent of the value, with only an implicit "leading one" for the mantissa. With exponent bits determining the range and the mantissa bits providing the accuracy, a delicate balance between the two is critical in optimizing a number representation for machine learning.

In this paper, we present the Adaptive Floating Point (AFP) representation, a denser alternative to IEEE FP32 or BFloat16 for storing ML data. AFP encapsulates floating point (FP) values into blocks of FP values along with characterization data that describe the block of numbers. AFP uses this block information to increase performance and reduce memory requirement while maintaining high accuracy.

AFP is designed to be effective in large-scale deep learning models. We use a collection of pre-trained benchmark models to analyze empirically the distribution of values obtained during inference, showing that 99% of values can be covered with AFP. In addition, previous compact representations for ML (Darvish Rouhani et al., 2020; Sa et al., 2018; Zhang et al., 2022) may require subsets of the calculations (e.g., certain layers) to be computed using high precision. Such a protocol may cause significant inefficiencies due to its data conversion and data transfer requirements. In contrast, AFP provides *end-to-end* application coverage by enabling *all* ML FP computations to utilize AFP.

Our paper's contributions include:

- We propose Adaptive Floating Point, a novel numerical datatype for use with *all* ML floating point data, as an alternative to IEEE-754 FP32 and Bfloat16 with higher memory and compute density. The key elements which provide higher density are:
  - *Auto Focus* feature, where a shared exponent and private offsets combine to automatically provide maximum available precision to 99% of all ML data.
  - Shared *block-level characterization* information to dynamically improve efficiency.
- We build a simulation infrastructure in Tensorflow to accurately model the numerical effects of applying AFP to the weights and layer outputs of ML models.

[1]Computer Science Department, Pomona College, Claremont, CA, USA [2]Computer Science Department, Occidental College, Los Angeles, CA, USA [3]Computer Science Department, University of California, Santa Cruz, CA, USA [4]Department of Computer Science, University of California, Irvine, CA, USA. Correspondence to: Thomas Y. Yeh <tomyenhsiyeh@gmail.com>.

Proceedings of the 39th International Conference on Machine Learning, Baltimore, Maryland, USA, PMLR 162, 2022. Copyright 2022 by the author(s).

- We perform comprehensive simulations of AFP on a wide range of robust CNN and Transformer models.

2. Background

The IEEE-754 single precision floating format (FP32) represents each value with one sign bit s, eight exponent bits e, and twenty-three mantissa bits m; this is illustrated in Figure 1a.

To increase both memory and compute density for ML applications, the BFloat16 format (Kalamkar et al., 2019) was introduced by Google. As shown in Figure 1b, BFloat16 retains the sign bit and the eight exponent bits while reducing the mantissa to seven bits. FP16 is another 16-bit format with a sign bit, five exponent bits, and ten mantissa bits.

For a vector of n elements, we can represent the elements in FP32, FP16 and BFloat16 representations with s, e, and m as the private signs, exponents, and mantissas:

$$\Big[(-1)^{s_0}2^{e_0}m_0, (-1)^{s_1}2^{e_1}m_1, ..., (-1)^{s_{n-1}}2^{e_{n-1}}m_{n-1}\Big].$$

In pursuit of ultra-low precision formats for training and inference, several custom mixed representations have been proposed (Chang et al., 2020; Liu et al., 2021; Yao et al., 2021), including IBM's RaPiD design which supports five different custom formats (Venkataramani et al., 2021).

An alternative approach to scaling the data formats is the block floating point (BFP) format. BFP have been used in signal processing platforms to optimize for both performance and memory density (Wilkinson, 1994; Kalliojarvi & Astola, 1996; Muller et al., 2010; Song et al., 2018). The Microsoft Floating Point (MSFP) (Darvish Rouhani et al., 2020), Hybrid Block Floating Point (HBFP) (Drumond et al., 2018) and FAST BFP (FAST) (Zhang et al., 2022) designs show the potential of this approach for ML applications. As shown in Figure 1c, MSFP, HBFP, and FAST approaches use a single, shared exponent (the largest exponent in the block) along with private sign and mantissa bits. Removing private exponent information both reduces the representation size of each value, and can also simplify the design of functional units.

In this work, we use BFP to refer generically to block floating point formats such as MSFP, HBFP, and FAST. One main disadvantage of BFP is the increasing loss of precision for values that are farther away from the maximum exponent, with smaller values eventually truncating to zero. MSFP, HBFP, and FAST designs have only been *selectively* applied to components of ML models in order to achieve accuracy. This can result in costly off-chip data transfers, especially for distributed systems, or slower execution on higher precision functional units.

For a given vector of n elements, BFP representations can be described with e_* as the shared maximum exponent, and s the m as the private sign and mantissa:

$$2^{e_*}\Big[(-1)^{s_0}\,m_0\,,\,(-1)^{s_1}\,m_1\,,\,...\,,\,(-1)^{s_{n-1}}\,m_{n-1}\Big].$$

3. Adaptive Floating Point

Our proposed AFP representation is designed for end-to-end application coverage where *all* ML FP computations are processed in AFP. AFP dynamically adjusts to the data to provide memory density while retaining model accuracy. In contrast to prior BFP formats, AFP is designed to maximize the benefit of encapsulating FP values into blocks. The structure of the AFP representation is shown in Figure 2.

For a given vector of n elements, the AFP representation is comprised of e_* as the maximum shared exponent, t as a private offset, and m as the private mantissa:

$$2^{e_*}\Big[(-1)^{s_0}\,2^{-t_0}\,m_0\,,\,...\,,\,(-1)^{s_{n-1}}\,2^{-t_{n-1}}\,m_{n-1}\Big].$$

The dot product of two given vectors with n elements $\vec{a}$ and $\vec{b}$ in the AFP format with shared exponents e_{a*} and e_{b*}, respectively, would take the form:

$$\vec{a}\cdot\vec{b}^{\,T} = 2^{e_{a*}+e_{b*}}\sum_{i=0}^{n-1}\Big((-1)^{s_{a,i}\oplus s_{b,i}}\,2^{-(t_{a,i}+t_{b,i})}\,m_{a,i}*m_{b,i}\Big),$$

where $\oplus$ is an XOR operation and T is the transpose.

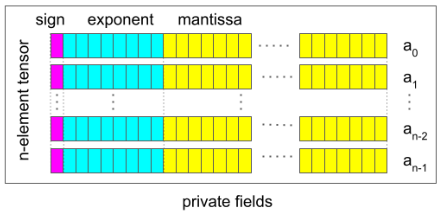
3.1. Design Details

AFP is a block based approach inspired by the BFP design from the signal processing domain and prior work for deep learning models. The representation contains two distinct sections of data: a private section with bit fields per value and a shared section with bit fields to describe the entire block. The AFP design described in the paper uses a 16-element block unless otherwise noted. The target accuracy for all rounding experiments in the paper is 99% of full precision FP32 accuracy.

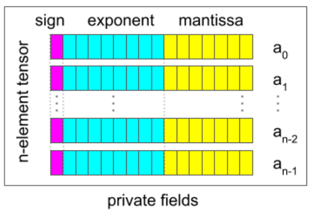
One key insight of AFP is to utilize characterization information about the bounded range described by the values in each block to increase the memory density of the data representation. For each component of the value (sign, exponent, and mantissa), AFP uses complementary private and shared fields.

3.2. Private Fields

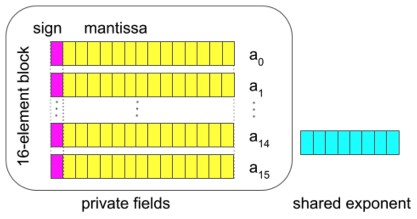
The private section includes 1 sign bit, 3 offset bits, and 5 mantissa bits. The sign bit represents the sign of the value.



(a) Single-precision IEEE



(b) BFloat16



(c) BFP

Figure 1. Previous FP representations used in machine learning. Compared to (a) single precision IEEE, (b) BFloat16 reduces the number of mantissa bits, while (c) existing block-based approaches such as MSFP, HBFP, and FAST group sequential blocks of fixed-point values and using a single, shared exponent with private sign and mantissa bits. This reduces the total number of bits, but may reduce the precision of elements that are farther from the maximum exponent value in the block.

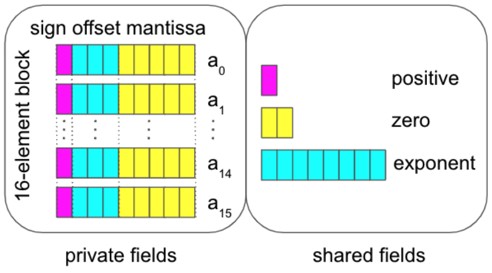


Figure 2. Adaptive Floating Point format (AFP). Compared to BFP, AFP uses private offset bits to enable using fewer mantissa bits while preserving fidelity. It also stores "positive" and "zero" shared fields to represent if a block is all positive and/or all not using one or more mantissa bits.

3.2.1. *Auto Focus* AND OFFSET

The offset bits indicate a particular value's offset from the maximum exponent of the 16-element block. If we have a block with a maximum exponent of 3, the value 2 would have an offset of 2 since 2^1 is 2 bit positions away from 2^3. One key observation is that the maximum exponent of a block dynamically focuses the range described by each block. AFP sacrifices a small number of mantissa bits (slightly reducing the precision to which all values are represented) in order to gain the same number of exponent bits, which significantly increases the precision of smaller values within the block. As we shall see in the experiments, this tradeoff enables high accuracy performance using far fewer bits in total. (All offsets are computed automatically by AFP's encoder and decoder hardware.) We call this feature *Auto Focus*.

Within any block, we can understand AFP's dynamic range in terms of the shared maximum exponent, e_*, the number of mantissa bits, n, and the number of offset bits, o. Then, the largest representable value is 2^{e_*+1}, while the smallest representable non-zero value is $2^{e_*-(n-1)-(2^o-1)}$. We can see that, for AFP with $o = 3$, $n = 5$, this lets us represent values as small as 2^{e_*-11}. In contrast, BFP with no offset and $n = 8$ gives smallest value 2^{e_*-7}. Thus, AFP can accommodate a much larger range in values within a block. This difference in dynamic range and available bits can also be seen in Figure 6.

Prior works that use fixed-point representations without blocks, such as IBM 4-bit (Sun et al., 2020), Mix and Match (Chang et al., 2020), and RaPiD (Venkataramani et al., 2021), increase the dynamic range and precision with programmable or trainable scaling factors. However, these techniques require significant developer investment in the form of manual calibration, hyper-parameter tuning, or model re-training. By employing a max exponent and a 3-bit private offset, AFP implicitly determines the appropriate range and quantization for each block.

3.2.2. 3-BIT OFFSET

Empirical analysis of the parameter values and intermediate computations of deep learning models shows the importance of incorporating our proposed 3-bit offset. Figures 3 and 4 show, respectively, the percentage of the model weights and layer outputs whose offset, from the shared maximal exponent value, is less than a given threshold. Using 3 offset bits, AFP can represent offsets ranging from 0 to 7.

Perhaps surprisingly, all the models exhibit very similar distributions; the average for each graph is shown in yellow. Across all the models we examined, 99% of all weights and layer outputs have offsets less than 8.

We hypothesize that the relatively tight grouping of values is common to trained models. For general scientific computing, we may need to accumulate many terms on different scales, some of which may cancel one another. In ML, the data is expected to be noisy, and the parameter values have been trained in order to provide accurate predictions on data from the training distribution. The data noise means that higher-scale values are unlikely to fully cancel, making the fidelity of lower-scale values less necessary to the overall calculation. Similarly, since the training process sets model parameters to optimize performance, the parameter values are likely to have relatively similar magnitudes to one another so that, on average over the data, they are able to influence the output prediction meaningfully.

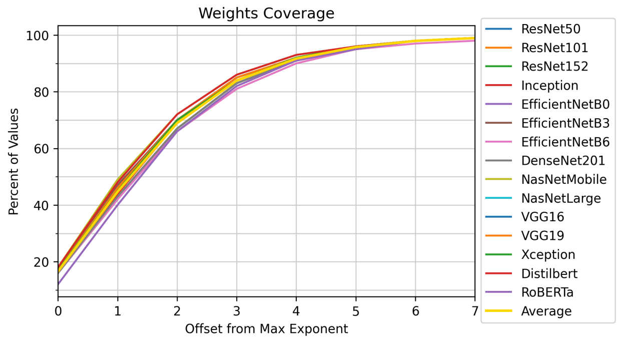


Figure 3. Percentage of weights representable by a given offset within 16-element blocks. While less than 20% of weights have the same exponent as the maximum, almost all (99%) are within offset 7, making a 3-bit private offset sufficient for preventing any precision loss in the mantissa.

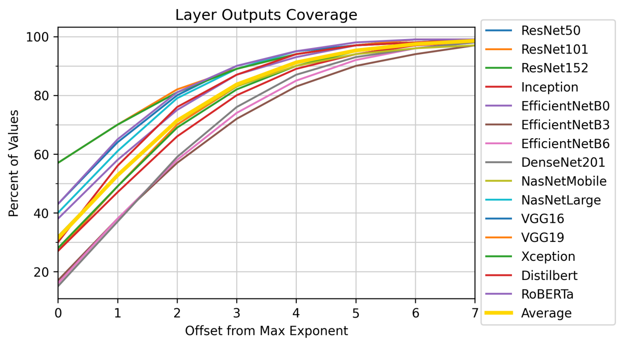


Figure 4. Percentage of layer outputs representable by a given offset within 16-element blocks. These values show more variability across models than the weights, but again, 3 private offset bits is sufficient to prevent precision loss on about 99% of values. On average, 19% of the layer outputs are zeros.

While we only show the data for 16-element blocks, a maximum offset of 7 still covers 99% of the values on average for 32-element and 64-element blocks. Based on our simulations, the fine grain block size of 16 provides a good balance between model accuracy and hardware optimization. Larger block sizes have minimal impact on memory density but increase hardware complexity and wire delays.

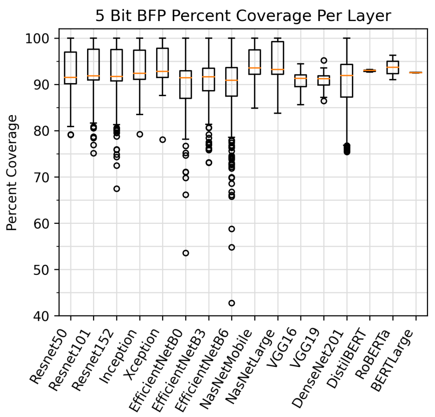
By providing the 3-bit offset with a range of 0 to 7, AFP provides its maximum available precision to 99% of all values. Prior block based FP formats without an offset result in significant data loss with insufficient mantissa bits. Values whose offset are greater than or equal to the number of mantissa bits are truncated to zero. Thus, BFP with (no offset and) 3, 4, or 5 mantissa bits respectively, captures only 69%, 85%, and 92% of values on average; in other words, 31%, 15%, and 8% of all values (respectively) are truncated to zero. In the sequel, we define *coverage* of a collection of values to be the fraction of those values that can be quantized by a representation without truncating them to zero. For 5 mantissa bits, BFP's coverage of the model weights appears adequate at 92%, but we can see a more complete picture by looking at *per layer coverage* of values.

To understand the per-layer coverage of weights, we show whisker plots comparing AFP to BFP using the same number of bits per value in Figure 5. Although the mean coverage is reasonable with 5 mantissa bits, we can see that many models contain layers that would undergo severe truncation to zero with BFP (as much as 57% of a layer's weights). The BFP plots 5a and 5b show many layers identified as low-coverage outliers (particularly the EfficientNet models), and even the non-outlier whiskers indicate many layers in each model with low coverage. For example, the minimum whisker value for BFP with 5 bits is 77%, meaning that 23% of all weights in these layers are truncated to zero. Even with 8 mantissa bits, BFP shows outliers with coverage as low as 55%.

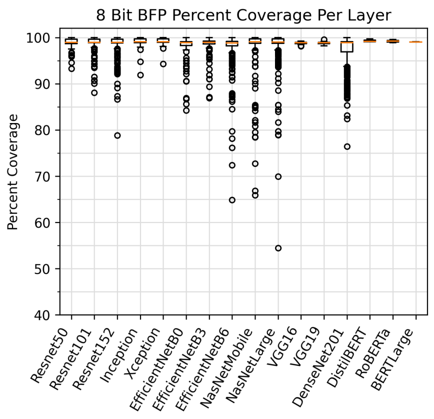
Since data inputs must propagate through each layer of a model, having multiple or even one highly truncated layer could severely degrade a model's accuracy. When using AFP, the 3-bit offset enables a significantly higher minimum of the coverage distribution. All layers' lower whiskers are at or above 98%, and the minimum outlier coverage is at 70%.

Even BFP values that are not completely truncated to zero may incur significant loss of precision if they are far from the maximum exponent. Again, in Figures 3 and 4, we see an average of 80% of weights and 70% of layer outputs will have one or more mantissa bits truncated. For a value with offset of 7, only 1 bit of precision is retained. Figure 6 illustrates this point by comparing AFP with BFP formats (MSFP, HBFP, and FAST) using 8 bits of private fields (excluding the sign bit) for each value. BFP use all 8 bits for the mantissa, while AFP uses a 3-bit offset and a 5-bit mantissa. The highlighted bit positions are the actual mantissa bits stored in each representation. Each row indicates the information stored for values with the particular offset from the maximum exponent.

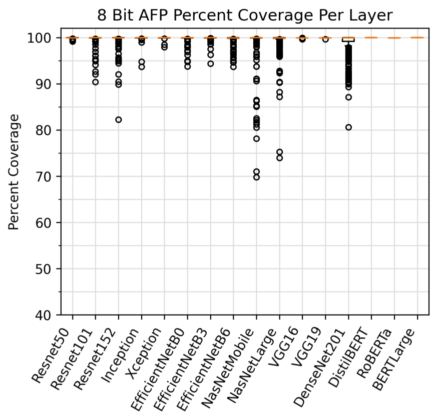
By leveraging each block's information, AFP can provide an additional bit of precision, indicated by the colored x's in Figure 6; the details of this expanded mantissa width are described in the next section. With AFP (left panel), all values with offsets up to and including 7 utilize the maximum amount of available precision. In contrast to BFP, AFP effectively stores the significant digits and provides a graceful degradation in precision all the way until bit 12.



(a) 5 Bit BFP Coverage



(b) 8 Bit BFP Coverage



(c) 8 Bit AFP Coverage

Figure 5. Whisker Plots of Per-Layer Percent Coverage Comparing AFP to BFP. BFP with 5 bits and 8 bits results in as much as 57% and 46% of a layer's values being truncated to zero. BFP's coverage applies to MSFP, HBFP, and FAST. AFP with 8 bits improves the coverage significantly, especially for the EfficientNet models.

3.2.3. MANTISSA

Compared to existing BFP formats, AFP's private offsets allow its 5-bit mantissa to behave more similarly to FP32's mantissa. In particular, for all values with offsets less than 7, an implicit leading one is concatenated to the 5-bit mantissa. For denormal values (offsets 7 or greater), no implicit one is used. This is illustrated on the left of Figure 6. The value zero is represented by an offset of $0b111$ and a mantissa of $0b00000$. Additionally, AFP uses nearest rounding on the least significant bit for inference instead of the truncation used by MSFP and HBFP. However, we plan to evaluate stochastic rounding of AFP data for the backward pass of ML training in future work.

The choice to use a 5-bit mantissa width and nearest rounding is motivated by ML inference simulation experiments, on which we compare three traditional rounding modes: nearest, truncate, and stochastic. To evaluate the role of the mantissa precision in model inference, we evaluate each model's accuracy when representing values using a full 8-bit private exponent, while varying mantissa lengths and rounding strategies. Intuitively, this evaluates the degradation associated with the mantissa size and rounding, absent any effects from the blocking or exponent range. Figure 7 shows that nearest rounding requires the fewest mantissa bits to achieve 99% of FP32 accuracy across all of the benchmarks. The maximum required mantissa width for nearest is 5 bits, while the maximum required mantissa width is 9 bits for truncate. We use round to nearest to achieve high accuracy with minimum number bits.

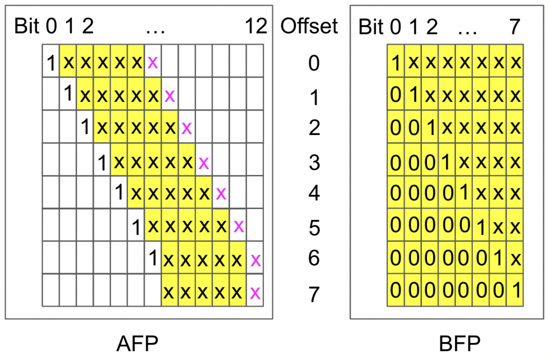


Figure 6. Comparing AFP and BFP with 8 Private Bits. AFP uses 3 offset and 5 mantissa bits, while BFP uses 8 mantissa bits. Yellow locations show which bits are explicitly stored in the mantissa. For higher-offset values, BFP quickly loses fidelity in its representation.

Intuitively, if each value preserves its exponent as in FP32, we still need a minimum of 5 mantissa bits to achieve the target accuracy across all benchmarks. In this sense, the 5-bit mantissa can be seen as a lower bound on the mantissa width for block based formats. As we show in subsequent experiments, AFP is able to achieve our target accuracy with a 5-bit mantissa and 3-bit offset.

Similarly, since MSFP and HBFP use truncation rounding, our experiment provides a lower bound on their required mantissa width, since our truncation uses a (more accurate) private exponent for each value instead of the shared exponent. In Figure 7, several models (specifically, the EfficientNet models) require a mantissa of 9 bits to maintain accuracy. While stochastic rounding has been used in many ML quantization papers (De Sa et al., 2017; Sa et al., 2018), our experiments showed its performance to be generally similar to truncate for inference.

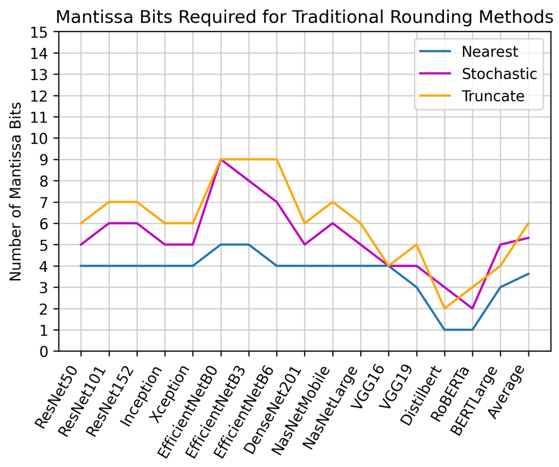


Figure 7. Mantissa Bits Required for Traditional Rounding. Round to nearest shows the best performance with a maximum of 5 bits and an average of 3.6 bits to achieve the target accuracy.

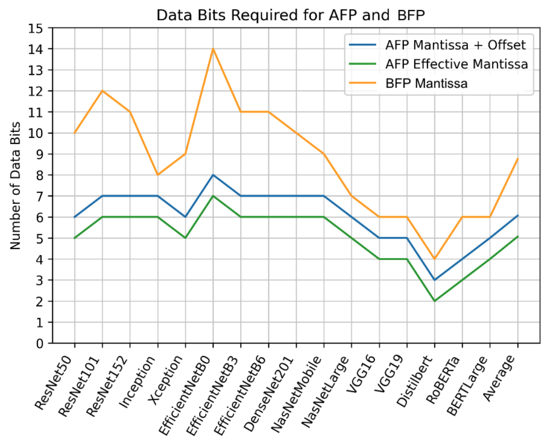


Figure 8. Data Bits Required to Achieve Target Accuracy. AFP's mantissa width is 1 bit less than the data width, and BFP's mantissa width equals its data width. The maximum required mantissa width is 7 for AFP and 14 for BFP. The average mantissa width is 5.1 for AFP and 8.8 for BFP.

3.3. Shared Fields

AFP's shared section uses 2 bytes to store information describing the entire block, including an 8-bit shared exponent and 8 bits of block characterization information. The 8-bit shared exponent is set to be the maximum exponent among the values within a block. We allocate a full byte for block characterization info to support byte alignment and enable future optimizations.

In this study, we explore two methods for improving efficiency. First, we include a field to indicate positivity. Second, we include an optional zero field for each block. These methods dynamically increase the mantissa bits by a maximum of one bit. By designing the AFP multiplier to use radix-4 multiplier encoding (Ercegovac & Lang, 2003), the extra bit can be supported with minimal overhead to the multiplier area and delay, since both 6 bit or 7 bit multiplier designs will generate 4 partial products to reduce.

3.3.1. Positive Field

The positive field is used to indicate if all values in a section are positive. While only one bit is necessary, two bits per block can be used to reduce the critical path and wire delay for more efficient hardware. The accuracy difference between using one vs. two positive bits is negligible. One bit is assigned to the first half of the block (8 values), and the second bit is assigned to the latter half. When a set of 8 values are all positive, the private sign bit is reused as an additional mantissa bit. While all-positive blocks are not common in the model weights (1% of blocks contain all-positive values), they are much more common in the layer outputs (35% of blocks), due to the popularity of ReLU and similar non-negative activations.

3.3.2. Zero Field

The zero field indicates bit positions where all values with the same offset have a zero following the leading one. The two bits describe values with an offset of 0 and 1. This shared information allows AFP to remove these *unused* bits from the stored mantissa, allowing an additional mantissa bit to be stored. The zeros are dynamically restored upon decode. To maximize compute density, the zero bits are ignored when the all positive bit is on. Our simulations show that zero bits increase the effective mantissa width for approximately 50% of the blocks. To simplify the hardware design, we can use 4 zero bits which are split into two 2-bit vectors, each referring to a set of 8 values.

3.4. Fixed Number of Bits

Many ML accelerators such as Google's TPU (Jouppi et al., 2021), Nvidia Tensor Cores, Intel NNP, Centaur Technology's Ncore (Henry et al., 2020), and Groq's TSP (Abts et al., 2020) are designed with a fixed data width. Figure 8 shows the required data width for each ML model. Each data format is applied to all weights and layer outputs for inference. Assuming a fixed width accelerator, to achieve our target performance requires choosing the maximum width across the various ML models. AFP requires a total of *8* data bits (the 3-bit offset and 5-bit mantissa), while BFP requires *14* mantissa bits. When comparing the total amount of memory to store a 16-element block, AFP increases memory density by 1.6×, 1.6×, and 3.2× over BFP, BFloat16, and FP32, respectively.

The square of effective mantissa width can be used as a first-order approximation of compute density. For AFP, the effective mantissa width in hardware is shown in Figure 8.

Model	Float32	BFloat16	AFP8
CNNs			
ResNet-50	1.000 (62.04)	1.000	0.999
ResNet-101	1.000 (64.05)	1.001	0.999
ResNet-152	1.000 (64.60)	1.000	0.998
Inception-v3	1.000 (66.56)	1.000	0.996
Xception	1.000 (67.72)	1.000	1.003
EfficientNetB0	1.000 (62.22)	1.002	0.994
EfficientNetB3	1.000 (69.36)	1.001	0.998
EfficientNetB6	1.000 (73.28)	1.000	0.997
DenseNet201	1.000 (65.56)	1.000	0.999
NasNetMobile	1.000 (61.67)	1.002	0.999
NasNetLarge	1.000 (72.21)	0.999	0.999
VGG16	1.000 (58.88)	1.000	0.999
VGG19	1.000 (58.82)	0.999	1.000
Transformers			
DistilBERT	1.000 (91.06)	1.000	1.000
RoBERTa	1.000 (89.96)	1.000	1.000
BERT-Large	1.000 (83.04)	1.000	1.000

Memory Density	1.0×	2.0×	3.2×
Compute Density	1.0×	9.0×	12×

Table 1. Accuracy and Density Normalized to FP32.

To achieve the target accuracy for all benchmarks, AFP requires *7* effective mantissa bits (only 5 stored), while BFP requires *14* bits. FP32 uses 24 mantissa bits, and BFloat16 uses 8 mantissa bits. When compared to BFP, BFloat16 and FP32, AFP increases compute density by 4×, 1.3× and 12×, respectively.

This enables AFP's accelerators to have 1.3× as many multiply-accumulate (MAC) units on chip as BFloat16. AFP's memory density also permits the same memory to store 1.6× as much data as BFloat16, and will have memory bandwidth effectively 1.6× greater than BFloat16. As described in Jouppi et al. (2021), memory bandwidth is a critical resource for maintaining high utilization of ML accelerators.

Detailed accuracy and density data normalized to the values for FP32 are shown in Table 1. Under the FP32 column, the raw accuracy percentage is shown for each benchmark inside the parentheses.

We use AFP8 to indicate the AFP representation with *Auto Focus* and shared positive bits only (i.e., no zero fields). The shared fields for AFP8 include the maximum exponent and positive bits while the private fields include 1 sign bit, 3 offset bits, and 5 mantissa bits. By using only the positive bits for accelerators with a fixed data width, AFP8 simplifies the hardware design.

Prior work show lower mantissa bit requirements for MSFP and FAST by only applying the BFP formats to certain components (e.g., convolutional layers only) of ML models, as well as performing model tuning (retraining).

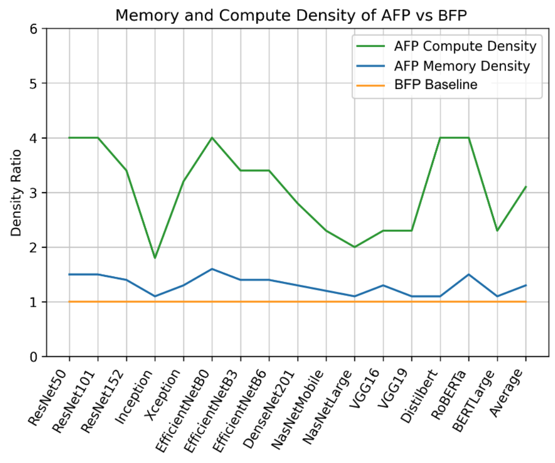


Figure 9. Comparing Compute and Memory Density of AFP vs BFP. With the same area and memory, employing AFP will allow 3.1x MAC units and 1.3x more values to be stored.

3.5. Variable Number of Bits

For ML accelerators that can utilize variable bit sizes such as FPGAs (Darvish Rouhani et al., 2020; Chang et al., 2020), Figure 8 shows each benchmark's minimum required data width for AFP vs. BFP. AFP's data width includes the private offset and the private mantissa. For BFP, the data width equals the private mantissa width.

Figure 9 compares the compute density and memory density of AFP vs. BFP. On average, memory density and compute density are increased by 1.3× and 3.1× when using AFP vs. BFP. With the same die area constraint, AFP enables 3.1× times as many MAC units to fit on the chip. AFP can store 1.3× as much data as BFP given the same amount of memory, and the memory bandwidth is effectively 1.3× that of BFP.

3.6. Run-time Analysis

We can estimate AFP's run-time improvement by using the Roofline Model (Williams et al., 2009). Depending on whether a model's speed bottleneck is memory bandwidth or computation, AFP should improve run-time by a factor similar to the memory density improvement or compute density improvement, respectively. If a model's run-time is bandwidth-limited, AFP improves over BFP by 1.6x. If the model is computation limited, then AFP improves over BFP by 4x.

3.7. Error Analysis

By combining the shared 8-bit exponent and private 3-bit offsets, AFP achieves a dynamic range of $(3.67e^{-40}, 3.38e^{38})$, which is comparable to both FP32 and BFloat16. The minimum non-zero value is represented with a shared exponent of -126, an offset of 7, and mantissa of zeroes followed by one in the lowest bit position. The maximum value is represented with a shared exponent of 127, an offset of 0, and mantissa of all ones. Zero is represented by an offset of 7 and all zeroes in the mantissa for any shared exponent.

For AFP, there are two sources of error when compared to FP32 and BFloat16. First, error is injected with the narrower mantissa width, which ranges from 6 to 7. Second, error is injected when values fall outside the representable range and are truncated to zero. However, this scenario happens for less than 1% of values on average, as shown in Figures 3 and 4.

For values with offset < 7, the upper bound of AFP's absolute error is $2^{(ep-n)}$, n is # of mantissa bits and ep is the value's private exponent, and is 1/33 or 3.03% for relative error. For values with offset ≥ 7, the upper bound is $2^{(e_*-(n+6))}$, e_* is the shared exponent.

Given the dynamic nature of AFP, we measured the absolute and relative error while executing all models. For weights, the average absolute error was 0.00013 and the average percentage error is 0.64%. For layer outputs, the average errors are 0.037 and 0.47%, respectively. Compared to BFP using the same amount of memory, AFP reduces absolute error by 23% and 46% for weights and layer outputs, respectively. AFP reduces relative error by 60% and 43% for weights and layer outputs, respectively.

3.8. AFP Hardware Design

Figure 10 shows a high level diagram of an AFP ML accelerator to calculate the dot product of two vectors. This design uses fused operations where multiple operations are applied to the data before encoding back to AFP or FP32. The design works similarly to BFP based representations, such as MSFP, HBFP, and FAST, with additional logic to handle AFP's offset bits.

The *Mem* block refers to the memory or register where AFP data is stored. A simple and fast decoder expands blocks of AFP data and sends the operands to the multipliers.

The private operand offsets for each product are added by simple 3-bit adders. Prior to the adder network for the dot product, each product is right shifted by this sum of offsets. The entire adder network operates in fixed-point. In parallel, per-block shared exponents are added to produce the result exponent for the entire block.

To further validate our design, we developed the AFP block decoder, the AFP block encoder, the FP32 encoder and the AFP Dot Product unit in logic design. For each AFP value, block decoding requires only three AND gates and one 2-1 Mux. The critical path is only increased by the delays of one AND gate and one 2-1 Mux.

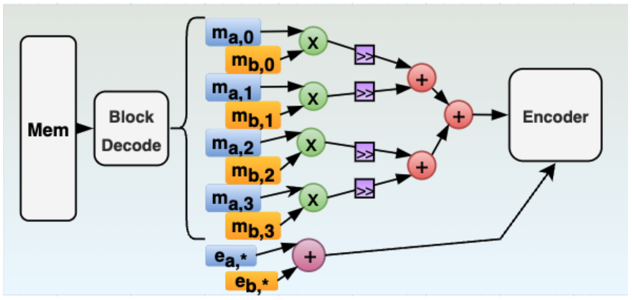


Figure 10. High Level Diagram of AFP Dot Product. The example uses a block size of 4. Multiplications occur with the decoded mantissas while private exponents $t_{a,n}$ and $t_{b,n}$ are added in parallel. The products are right shifted by the sum of offsets before reaching the adders. The entire adder network is in fixed point. The shared block exponents are used for encoding to AFP8 or FP32.

4. Experimental Setup

To simulate AFP in hardware, several DNN inference models were employed in Tensorflow using the Keras (Chollet et al., 2015) and HuggingFace (Wolf et al., 2019) libraries. For image classification, the test dataset of ImageNetV2 (Recht et al., 2019) was used to measure model accuracy[1]. The models used include ResNet-50, ResNet-101, ResNet-152, Inception-v3, Xception, EfficientNetB0, EfficientNetB3, EfficientNetB6, DenseNet201, NasNetMobile, NasNetLarge, VGG16, and VGG19. For transformer-based models, we applied AFP to DistilBERT for sentiment analysis (SST-2 dataset), RoBERTa for textual entailment (MNLI dataset), and BERT-Large for question-answering (SQUAD dataset).

Using a custom round function, all types of layers' weights and outputs were rounded with AFP, such as Conv2D, Batch Normalization, and Dense layers. To properly simulate inference using AFP in hardware, all the weights were rounded when instantiating the model and all layer outputs were rounded between every layer, before being input into the next layer. Examples from data sets were individually input into each model and a block size of 16 was used for most experiments. Entire data sets were always used to determine accuracy values.

To round values to the AFP format, we employ a method similar to Kalamkar et al. (2019) where FP32 operations emulate the behavior of AFP operations by zeroing out the lower mantissa bits outside of the AFP mantissa range. This rounding framework is built on top of Tensorflow, and we utilize the same infrastructure to model the other rounding methods described in this paper.

Because AFP is block based, we flatten and reshape the

[1]Note that the validation in Darvish Rouhani et al. (2020) used ImageNetV1; here we use V2 to provide a more realistic data distribution for evaluation.

tensors into blocks of 16 elements in order to accurately model the effects of AFP encoding. The maximum exponent is determined for each block, and all values' private offset is generated based on the maximum exponent. At the end of the rounding process, we reshape the tensors back into their original shapes. For tensors that do not divide evenly into 16-element blocks, we pad the end of the tensor with zeros and remove these padded zeros at the end of rounding. Across all the models, memory increase from zero padding is negligible, 0.0001% on average for weights and layer outputs.

5. Related Work

There are a number of well-known works that have proposed quantization or low-precision representations for accelerating machine learning, among them BFloat16 (Kalamkar et al., 2019), Nvidia Mixed Precison (Micikevicius et al., 2017), Buckwild (De Sa et al., 2017), High-Accuracy Low Precision Training (HALP) (Sa et al., 2018), IBM 4-bit (Sun et al., 2020), Microsoft Floating Point (MSFP) (Darvish Rouhani et al., 2020), and RaPiD (Venkataramani et al., 2021) This section will provide additional background on these representations.

Compared to FP16, BF16 increases the dynamic range but reduces precision. As discussed above, AFP improves upon the memory and compute density over BF16. Nvidia Mixed Precision makes use of IEEE half precision FP16 and FP32. It saves a master copy of the weights in FP32 to update weights during training and reduce compute time.

Stanford Buckwild uses a fixed point INT8 format focused on stochastic gradient descent (SGD) network training. Due to a fixed point format having a global exponent, it suffers from a narrow dynamic range. HALP also focuses on SGD training with a fixed point representation but employs a bit centering method which routinely narrows the representable range ,as gradient values converge, in order to increase precision. IBM 4-bit uses a 4-bit (1,3,0) exponent only representation to maximize the dynamic range for gradients and an INT8 quanitzation for performance-critical layers. It also employs per layer trainable scaling factors known as GradScale to decrease quantization error. GradScale is inconvienient to apply to existing models because the scaling factors require further training.

Mix and Match quantizes weights matrices row-by-row by employing a 4-bit fixed point representation with a scaling factor for uniformly distributed rows and a sum-of-power-of-2 (SP2) representation with scaling factor for gaussian distributed rows. Narrow fixed point representations like INT4 can be used in conjunction with loss scaling to train low-precision models, but is not effective at quantizing already-trained models. Unlike Mix and Match, AFP uses a private 3-bit offset which automatically adjusts to all distributions with greater precision. Also, AFP applies to 16-element blocks instead an entire row of data.

MSFP implements a bounding box method which is similar to AFP blocks. It also implicitly determines a max exponent. However, MSFP uses fixed-point values which may poorly represent non-uniform box distributions. Additionally, MSFP has only been applied to performance-critical layers in addition to model fine-tuning to achieve high accuracy. With the 3-bit offset, AFP can represent distributions with greater precision and can be applied to all layers of models without model fine-tuning.

FAST implements a dynamic precision system on top of the BFP format for DNN training. Similar to MSFP, the BFP format is applied to selective layers within the 6 models studied. FAST utilizes a 3-bit shared exponent with some form of scaling factor.

RaPiD applies a spectrum of precisions from FP16 to 2 bit fixed point for training and inferencing. For training, Hybrid-FP8 uses a (1,4,3) scheme for activations and weights and (1,5,2) representation for gradients. For inferencing, it uses PArameterized Clipping acTivation (PACT) and Statistics-aware Weight Binning (SaWB) with scaling factors to represent activations and weights. RaPiD may increase software and hardware complexity from the programmable scaling factor and the use of several different representations.

6. Conclusions

In this paper, we present AFP, an adaptive floating point representation for ML applications. AFP provides a highly compact format while maintaining the necessary fidelity to preserve accurate inference across a wide range of benchmark networks.

Ongoing work includes developing hardware implementations for AFP, as well as evaluating the effectiveness of AFP for model training. Prior work (Jouppi et al., 2021; Venkataramani et al., 2021) suggests that training may require more dynamic range than inference; however, we expect AFP to improve on existing representations in these settings as well, due to its increased ability to retain precision.

Acknowledgements

We would like to thank the anonymous reviewers for their helpful comments, Milos D. Ercegovac for early discussions on BFP, Kim B. Bruce for valuable feedback, and Anthony J. Clark for simulation discussions. We also want to acknowledge Carl Bell, Ziang Xue, and other members of the ARCALA Research Lab for assistance on model simulations and hardware design.

References

Abts, D., Ross, J., Sparling, J., Wong-VanHaren, M., Baker, M., Hawkins, T., Bell, A., Thompson, J., Kahsai, T., Kimmell, G., Hwang, J., Leslie-Hurd, R., Bye, M., Creswick, E., Boyd, M., Venigalla, M., Laforge, E., Purdy, J., Kamath, P., Maheshwari, D., Beidler, M., Rosseel, G., Ahmad, O., Gagarin, G., Czekalski, R., Rane, A., Parmar, S., Werner, J., Sproch, J., Macias, A., and Kurtz, B. Think fast: A tensor streaming processor (TSP) for accelerating deep learning workloads. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 145–158, 2020.

Chang, S., Li, Y., Sun, M., Shi, R., So, H. K., Qian, X., Wang, Y., and Lin, X. Mix and match: A novel FPGA-centric deep neural network quantization framework. *CoRR*, abs/2012.04240, 2020. URL https://arxiv.org/abs/2012.04240.

Chollet, F. et al. Keras. https://keras.io, 2015.

Darvish Rouhani, B., Lo, D., Zhao, R., Liu, M., Fowers, J., Ovtcharov, K., Vinogradsky, A., Massengill, S., Yang, L., Bittner, R., Forin, A., Zhu, H., Na, T., Patel, P., Che, S., Chand Koppaka, L., SONG, X., Som, S., Das, K., T, S., Reinhardt, S., Lanka, S., Chung, E., and Burger, D. Pushing the limits of narrow precision inferencing at cloud scale with Microsoft floating point. In *Advances in Neural Information Processing Systems*, volume 33, pp. 10271–10281, 2020.

De Sa, C., Feldman, M., Ré, C., and Olukotun, K. Understanding and optimizing asynchronous low-precision stochastic gradient descent. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pp. 561–574, 2017.

Drumond, M., Lin, T., Jaggi, M., and Falsafi, B. Training dnns with hybrid block floating point. In *Advances in Neural Information Processing Systems*, pp. 451–461, 2018.

Ercegovac, M. D. and Lang, T. *Digital Arithmetic*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2003. ISBN 1558607986.

Henry, G., Palangpour, P., Thomson, M., Gardner, J. S., Arden, B., Donahue, J., Houck, K., Johnson, J., O'Brien, K., Petersen, S., Seroussi, B., and Walker, T. High-performance deep-learning coprocessor integrated into x86 SoC with server-class CPUs industrial product. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 15–26, 2020.

Jouppi, N. P., Hyun Yoon, D., Ashcraft, M., Gottscho, M., Jablin, T. B., Kurian, G., Laudon, J., Li, S., Ma, P., Ma, X., Norrie, T., Patil, N., Prasad, S., Young, C., Zhou, Z., and Patterson, D. Ten lessons from three generations shaped Google's TPUv4i: Industrial product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pp. 1–14, 2021.

Kalamkar, D., Mudigere, D., Mellempudi, N., Das, D., Banerjee, K., Avancha, S., Vooturi, D. T., Jammalamadaka, N., Huang, J., Yuen, H., Yang, J., Park, J., Heinecke, A., Georganas, E., Srinivasan, S., Kundu, A., Smelyanskiy, M., Kaul, B., and Dubey, P. A study of BFLOAT16 for deep learning training, 2019.

Kalliojarvi, K. and Astola, J. Roundoff errors in block-floating-point systems. *IEEE Transactions on Signal Processing*, 44(4):783–790, Apr 1996. ISSN 1941-0476. doi: 10.1109/78.492531.

Liu, X., Ye, M., Zhou, D., and Liu, Q. Post-training quantization with multiple points: Mixed precision without mixed precision. *arXiv:2002.09049 [cs, stat]*, Jan 2021. URL http://arxiv.org/abs/2002.09049. arXiv: 2002.09049.

Micikevicius, P., Narang, S., Alben, J., Diamos, G. F., Elsen, E., García, D., Ginsburg, B., Houston, M., Kuchaiev, O., Venkatesh, G., and Wu, H. Mixed precision training. *CoRR*, abs/1710.03740, 2017.

Muller, J.-M., Brisebarre, N., de Dinechin, F., Jeannerod, C.-P., Lefèvre, V., Melquiond, G., Revol, N., Stehlé, D., and Torres, S. *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2010.

Recht, B., Roelofs, R., Schmidt, L., and Shankar, V. Do imagenet classifiers generalize to imagenet? In *International Conference on Machine Learning*, pp. 5389–5400, 2019.

Sa, C. D., Leszczynski, M., Zhang, J., Marzoev, A., Aberger, C. R., Olukotun, K., and Ré, C. High-accuracy low-precision training, 2018. URL http://arxiv.org/abs/1803.03383.

Song, Z., Liu, Z., Wang, C., and Wang, D. Computation error analysis of block floating point arithmetic oriented convolution neural network accelerator design. In *AAAI*, 2018.

Sun, X., Wang, N., Chen, C.-Y., Ni, J., Agrawal, A., Cui, X., Venkataramani, S., El Maghraoui, K., Srinivasan, V. V., and Gopalakrishnan, K. Ultra-low precision 4-bit training of deep neural networks. In *Advances in Neural Information Processing Systems*, volume 33, pp. 1796–1807, 2020.

Venkataramani, S., Srinivasan, V., Wang, W., Sen, S., Zhang, J., Agrawal, A., Kar, M., Jain, S., Mannari, A., Tran, H., Li, Y., Ogawa, E., Ishizaki, K., Inoue, H., Schaal, M., Serrano, M., Choi, J., Sun, X., Wang, N., Chen, C.-Y., Allain, A., Bonano, J., Cao, N., Casatuta, R., Cohen, M., Fleischer, B., Guillorn, M., Haynie, H., Jung, J., Kang, M., Kim, K.-h., Koswatta, S., Lee, S., Lutz, M., Mueller, S., Oh, J., Ranjan, A., Ren, Z., Rider, S., Schelm, K., Scheuermann, M., Silberman, J., Yang, J., Zalani, V., Zhang, X., Zhou, C., Ziegler, M., Shah, V., Ohara, M., Lu, P.-F., Curran, B., Shukla, S., Chang, L., and Gopalakrishnan, K. Rapid: Ai accelerator for ultra-low precision training and inference. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pp. 153–166, 2021. doi: 10.1109/ISCA52012.2021.00021.

Wilkinson, J. H. *Rounding Errors in Algebraic Processes*. Courier Corporation, Jan 1994.

Williams, S., Waterman, A., and Patterson, D. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65—-76, April 2009.

Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., Davison, J., Shleifer, S., von Platen, P., Ma, C., Jernite, Y., Plu, J., Xu, C., Scao, T. L., Gugger, S., Drame, M., Lhoest, Q., and Rush, A. M. Huggingface's transformers: State-of-the-art natural language processing, 2019. URL https://arxiv.org/abs/1910.03771.

Yao, Z., Dong, Z., Zheng, Z., Gholami, A., Yu, J., Tan, E., Wang, L., Huang, Q., Wang, Y., Mahoney, M. W., and Keutzer, K. HAWQV3: Dyadic neural network quantization. *arXiv:2011.10680 [cs]*, Jun 2021. URL http://arxiv.org/abs/2011.10680.

Zhang, S. Q., McDanel, B., and Kung, H. FAST: DNN training under variable precision block floating point with stochastic rounding. In *28th IEEE International Symposium on High-Performance Computer Architecture (HPCA-28)*, 2022.