Study of Posit Numeric in Speech Recognition Neural Inference

Zishen Wan, Eric Mibuari, En-Yu Yang, Thierry Tambe
Harvard University
Cambridge, MA

ABSTRACT

Recurrent neural networks (RNNs) are now at the heart of many applications such as image captioning, speech recognition, language translation and modelling. In order to achieve the best possible inference accuracy, these RNN models tend be quite large and many utilize 16 or 32-bit floating point format, which can enormously stretch memory resources. Quantizing the model parameters in lower bit precision would be required in order to deploy these models to edge and mobile devices with limited memory storage. In this report, we study the impact of various bit precisions and formats on speech-to-text neural performance. A particular emphasis is given to the posit numerical format which can yield higher accuracy than floats for a specific range of numbers. For this purpose, we implemented a Python module which can convert float numbers into posits of various bit lengths. Our speech recognition inference results show that the posit numerical format is by far the best solution for aggressive quantization at 8-bit. Furthermore, we show that the area and power cost of a posit-based hardware is smaller compared to a fixed-point-based hardware. Finally, we designed a hardware prototype design of a processing engine for speech-to-text inference in SystemC and successfully verified its functionality.

Index Terms – recurrent neural networks, low bit precision, posit format, speech recognition, word error rate, RNN hardware accelerator

1 INTRODUCTION

Recently, recurrent neural networks have produced very high performance across a variety of natural language processing (NLP) tasks. RNNs are now the state-of-art solution for speech recognition and have achieved remarkably low word error rates [7]. Moreover, advances in neural machine translation have narrowed the performance gap versus human translators [16]. The sequence-to-sequence (seq2seq) deep learning architecture [6], shown in Figure 1, has been the engine behind a lot of this progress in both speech recognition and machine translation. The encoder or *listener* receives the audio spectrograms or the text embeddings as input. And the decoder or *speller* outputs the translated text or audio captions. The outputs of the encoder and the inputs to the decoder are fed to a multilayer perceptron (MLP) feed-forward network in order to provide longer context between words. This is known as the attention mechanism [4] wherein a weighted combination of all the encoder outputs is computed into a context vector which gets consumed by the decoder. As this sequence-to-sequence model does not assume conditional independence between predictions, the attention mechanism helps the network learn an implicit language model from the training corpus and can therefore optimize the word error rate (WER) more directly than other acoustic models relying on an external language model for decent performance.

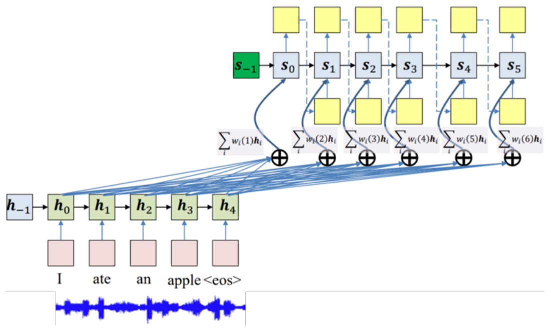


Figure 1: Sequence-to-sequence model for translation or speech recognition

However, it is worth pointing out that an external language model can still be plugged into this attention-based network for increased predictive accuracy [5].

A high performance attention-based seq2seq speech recognition model can easily surpass 30M parameters, consuming 120MB of memory in native 32-bit precision, which is well beyond the memory storage envelope of a typical system-on-chip. To alleviate this challenge, techniques such as sparse execution [10] and low bitwidth quantization [15] have been proposed in order to compress the neural network without hurting performance. In this work, we are particularly interested in quantizing the parameters of seq2seq speech recognition models in low posit bit precision. The posit data type, belonging to a larger category of numerics called unums, can offer compelling advantages in dynamic range and accuracy over IEEE standard 754 floating-point type [9]. We are especially motivated in the observation that the decimal accuracy of the posit number system is greater compared to floats in the dynamic range of values commonly seen in trained neural network parameters. Therefore, the main goal of this work is to study the efficacy of the posit number system for speech-to-text inference. In doing so, this report makes the following contributions:

- Develop a Python-based framework for converting in all directions between float, fixed-point and posit numbers.
- Compare and evaluate the efficacy of the posit type as well as many common numerical data types in neural speech recognition inference.
- Demonstrate that the hardware cost of a posit-based hardware can be smaller in terms of area and power compared to a fixed-point or standard floating-point based hardware.
- Unveil a hardware prototype design of a processing engine for speech-to-text inference.

2 OVERVIEW OF COMMON NUMERICAL DATA TYPES

In this work, we considered a number of common and popular numerical data types for speech-to-text inference besides posit, mainly: IEEE754 32-bit and 16-bit floating point, 16-bit bfloat, 16-bit and 8-bit fixed-point as depicted in Figure 2.

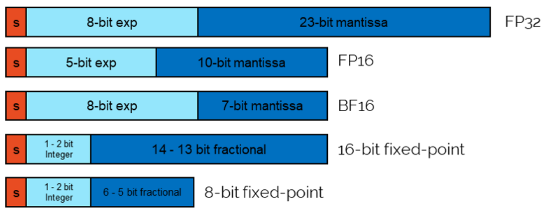


Figure 2: Common numerical data types used in Today's computers and ASICs

In particular, the bfloat16 number format preserves the dynamic range of 32-bit float by retaining eight exponent bits but incurs reduced precision with 7 fractional bits. Bfloat16 has garnered popularity as of late and is currently utilized in 2^{nd} generation Google cloud TPUs as well as in Intel AI processors and FPGAs [1]. Moreover, low precision fixed-point arithmetic have become the standard for performing aggressively-compressed deep learning inference [8, 12]. In the next section, we detail the anatomy of the posit number format which will be used and contrasted against the above-mentioned popular data types for neural speech recognition.

3 THE POSIT NUMBER SYSTEM

Efficient hardware acceleration especially for edge devices requires multiple approaches to achieve reductions in power and memory use. Word bit length reduction is one key way of achieving this objective. One of the most promising innovations, that helps with word bit length reduction, that has recently received attention and that we have explored in our project is the use of Posit numerics. Use of posits promises reductions in data movement costs, reduced latency and reduced power dissipation. The numbers that are used in neural networks especially parameters such as weights are especially suitable for representation in posit format. In our project we demonstrate the achievements of these objectives with an 8-bit word length, but in addition, we demonstrate a flexible word-length implementation, which to the best of our knowledge has not been done before.

3.1 Introduction to Posits

Posits belong to a larger category of number formats called unums that were invented by John Gustafson [9]. Posits closely resemble floating point numbers in format because they also round off the number to the nearest expressible value if the result of a calculation is not expressible exactly, but there are two major differences. For the same number of bits:

- Posit offers more accuracy than floats
- Posit offers a large dynamic range than floats

The better accuracy and more dynamic range are achieved by better rounding rules, more flexible use of bits available, and fewer exceptions. In general, in the posit representation, numbers with a smaller exponents are represented more accurately compared to numbers with large exponents. This is because the exponent has approximately Gaussian distribution.

3.2 The Posit Format

The equation below shows how a posit number is represented.

$$x = \begin{cases} 0, & \text{if } (00...0). \\ NaR, & \text{if}(10...0) \\ (-1)^s \times 2^{2^{es} \times k} \times 2^e \times (1 + \frac{f}{2^{fs}}), & \text{otherwise.} \end{cases} \quad (1)$$

Where:

s represents the sign. 0 is used for positive numbers and 1 for negative numbers

es represents the maximum number of bits allocated for the exponent

fs represents the maximum number of bits allocated for the fraction

e is the exponent value

f is the fraction value

k is computed using the equation below

$$k = \begin{cases} -m, & \text{if r = 0.} \\ m + 1, & \text{if r = 1} \end{cases} \quad (2)$$

Where: m is the length of the regime bits, and r is the bit (either 0 or 1) used in the regime. The regime is the length of identical bits that immediately follow the sign bit, and are terminated by the opposite bit, called the regime terminating bit, or the end of the n-bit value.

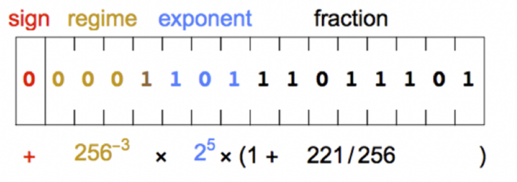
The posit format equations above shows two exceptional cases:

(1) To represent the zero, we use a single bit pattern, a continuous string of only zeroes (00 ... 0). There"s no requirement to distinguish between negative and positive zeros as is done by floats.
(2) To represent Nans and infinity, the singe bit patter (10 ... 0) a one followed by a string of zeroes is used. NaR stands for "Not a Real" and is used to represent exception values and infinity.

The component $2^{(2^{es})}$ is called the **useed** and is used quite frequently in the computations to convert back and for the between Posit and other formats.

3.3 Example of posit format representation

The example below shows a sample posit bit string (Gustafson, 2017). Assume we are told that the es environment variable is 3.



In this example:

- The sign bit is 0, meaning we are representing a positive number.
- The regime bits are the three 0's terminated by the opposite 1. Therefore, the value used in the useed exponentiation is -3.
- The exponent bits are 101, equivalent to 5 which is an unsigned binary integer
- The fraction bits are 11011101, equivalent to 221, also an unsigned binary integer. When this fractional part is converted to a fraction using the useed we get $1 + \frac{221}{256}$

Therefore, the bit string shown above evaluates to:

$$+256^{-3} \times 2^5 \times \left(1 + \frac{221}{256}\right) = 3.55393 \times 10^{-6}$$

3.4 Conversion from real numbers to posit

The posit environment is specified using two numbers: N - the number of bits in the word, and es - the exponent scale (the maximum number of bits allocated to the exponent).

If the numbers are any of the exceptions, shown in equation 1, we return the appropriate exception string.

If not, the conversion algorithm from real to posits depends on where the absolute value of the number lies on the number line. There are three main categories of numbers:

(1) Those whose absolute value is greater than useed
(2) Those whose absolute value lies between 1 and useed
(3) Those whose absolute value lie between 0 and 1

The sign bit is simply read from the number and the added at the beginning of the posit string. Otherwise, the rest of the conversation only depends on the absolute value of the real number being converted.

3.4.1 Real number greater than useed. For illustration purposes, we first show the conversion for numbers in category 1 above. The algorithm for conversion of reals to posit is as follows:

```
(1) Compute the useed as $2^{2^{es}}$
(2) Get the regime bits
    (a) Compute regime length - how many times useed fits whole
        in the real number
    (b) Convert regime length (k) to bits as per equation 2
(3) Get the exponent bits
    (a) Divide original real number by the $useed^k$ to get the exponent and fraction component
    (b) Get the base 2 exponent of this exponent and fraction
        component
    (c) Convert this exponent to exponent bits
    (d) Adding padding if necessary to reach the exponent length
        es
(4) Get fraction bits
    (a) Divide the exponent and fraction component by the exponent component ($2^{\text{exponent}}$)
    (b) Result is the mixed fraction
    (c) Subtract one from mixed fraction to obtain fraction component
    (d) Convert fraction component to binary = fraction bits
(5) Set posit string = sign bit + regime bits + exponent bits +
    fraction bits
(6) Return posit string
```

3.4.2 Real number between 1 and useed. In the second category where the absolute value of the number lies between 1 and useed, the conversion algorithm is as follows:

In this case, we'll not really use the regime part of the bit word. However, we still need to use at least two bits for the regime, one to indicate that our regime exponent is zero, and the other to show the regime termination. That is, the scale factor coming out of the regime part becomes 1 (exponentiation by zero).

Instead of steps 1 and 2 of the algorithm above, we set the regime bits to "01". The real number is then used as the exponent and fraction component that we get in step 3(a) of algorithm 3.4.1 above, and the rest of the algorithm resembles algorithm 3.4.1.

3.4.3 Real number less than 1. In this case:

```
(1) Find the combined exponent for the useed and the base 2
    exponent
(2) Split up the combined exponent into useed power and base
    2 exponent power
(3)(a) Determine useed power
   (b) Determine exponent power
(4) Set posit string = sign bit + regime bits + exponent bits +
    fraction bits
(5) Return posit string
```

3.5 Conversion from posit to real numbers

The conversion from posit to real numbers primarily involves parsing the string of posit bits, and picking out the different components for the sign bit, regime bits, exponent bits and fraction bits and then multiplying the different factors.

(1) Check if the posit string matches one of the exception cases i.e. if it's a string of all zeroes or a one followed by zeroes which would match the real values zero and infinity respectively.
(2) The es value is an environmental parameter that is read from the settings
(3) Read the sign of the real number by reading the first bit of the string
(4) Determine the lengths of the regime by reading the number of unbroken sequence of similar bits after the sign bit. This also determines the sign of the regime power (i.e. whether negative or positive), depending on whether the regime is a string of zeroes or ones. Hence the regime exponent is determined
(5) The exponent bits follow the regime bits and the regime terminating bit. The length of the exponent bits is es, so it's easy to know where the exponent sequence ends.
(6) Determine the exponent value using good-old binary to integer conversion of the exponent bits sequence
(7) The remaining bits represent the fractional part. These are then converted to their real value by using a binary to decimal conversion for a fractional value.

(8) The final real number is determined by multiplying the various scale factors as determined from the regime component, the exponent component and the fractional part.

3.6 Posit with flexible bit numbers

The most efficient state of the art posit libraries (such as SoftPosit from Berkeley) have implemented posit for a limited set of common bit lengths: 8, 16 and 32. One of the major contributions of our work is the implementation of posit for word lengths of any size. In our implementation the length of the word is supplied as an environment parameter, just like the es value is supplied by previous implementations. This added flexibility means that our implementations can be deployed in uses cases where the word lengths are not known a-priori.

4 RESULTS

Using the OpenNMT toolkit [11], we trained two attention-based seq2seq models for speech recognition on the open-source LibriSpeech corpus [14]. The structural architecture of these two models generally follows DeepSpeech3 specifications [5]. But we substituted batch normalization for layer normalization [3] because we found that it worked best in this application.

The first model, which uses MLP attention [13], contains 4 layers of bidirectional GRU with 1024 hidden units and 2 downsampling pooling layers in the encoder and a 1-layer forward-only decoder with 512 hidden units. This network, referred as Model 1, has about 20M parameters and was trained to a WER of 18.80 in native 32-bit floating-point type.

The second model employs general attention [13] and contains 5 layers of uni-directional LSTM with 800 hidden units and 4 downsampling pooling layers in the encoder and a 2-layer forward-only decoder with 512 hidden units. This network, referred as Model 2, has about 30M parameters and was trained to a WER of 26.28 in 32-bit floating-point precision.

Table 1 summarizes the efficacy of quantizing the model's parameters into various numerical types and bidwidth precisions on both seq2seq models. Note that the networks were not retrained after quantization was applied. Two main observations can be drawn from these results.

The first observation is that 8-bit posit is the best numerical solution for aggressive quantization at 8-bit precision. Model 1 incurs only a 0.3 percentage point loss in the WER accuracy and Model 2 shows no degradation when 8-bit posit is used as data type. Whereas, 8-bit fixed-point precision yields a much more appreciable accuracy loss in both models.

The second observation is that all 16-bit numerics (FP16, BF16, 16-bit posit, 16-bit fixed-point) have similar performance on both models and match the performance of the native IEEE754 FP32. This is to be expected given that the distribution of the weights is [-2.5, 2.5] for Model 1 and [-2, 2] for Model 2. Therefore, the increased fractional precision of these 16-bit numerics is big enough to preserve the performance of the native FP32 data type. Interestingly, the 8-bit fixed-point<2,6> type in Model 1 shows higher accuracy than the 8-bit fixed-point<3,5> type even though, in the former type, there is not enough integer bit width to represent numbers larger than 2. This suggests that there is a denser weight distribution in the [-2, 2] range while the distribution is sparser beyond -/+2.

Table 1: Impact of quantizing the model's parameters into Posit and other popular data types on speech recognition performance during inference

Data type and Bitwidth	WER of Model 1	WER of Model 2
Native (IEEE754 FP32)	18.80	26.28
8-bit fixed-point<2,6>	22.33	27.25
8-bit fixed-point<3,5>	37.05	-
8-bit posit<8,0>	**19.10**	**26.28**
IEEE754 FP16	18.80	26.29
Bfloat16	18.97	26.36
16-bit fixed-point<2,14>	18.80	26.27
16-bit fixed-point<3,13>	18.78	-
16-bit posit<16,1>	**18.80**	**26.28**

Finally, the small WER deterioration when using 8-bit posit quantization in Model 1 was recovered after just 1 epoch of supervised training and posit re-quantization. The WER of the 8-bit posit quantized model improved from 19.10 to 18.84 after re-training.

5 AREA AND POWER COSTS OF POSIT IN HARDWARE

Figure 3 shows the concept of posit-based SoC. The weights and dynamic activations of the DNN model are stored in the main on-chip memory in posit format. The accelerator performs RNN computations in posit-based arithmetics. Therefore, we need a posit-based adder, multiplier and accumulator for use in the processing engine of the accelerator.

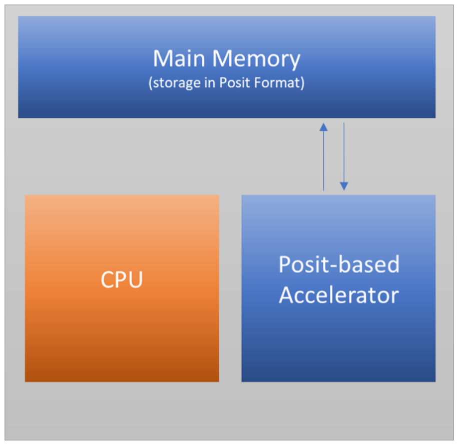


Figure 3: Framework of posit-based SoC

We achieve the transformation between posit number, floating-point number and fixed-point number, as well as MAC posit operations using C++, which can be combined with LSTM engine to form a posit-based accelerator. For the accuracy measurement method, considering the relative error formula is quite different for numbers and their converses, we adopt decimal error:

$$\text{decimal error} = |\log_{10}(\frac{x_{\text{computed}}}{x_{\text{exact}}})|$$

The decimal error can be used to define decimal accuracy. Accuracy is the inverse of error.

$$\text{decimal accuracy} = -\log_{10}|\log_{10}(\frac{x_{\text{computed}}}{x_{\text{exact}}})|$$

For 8, 16 and 32 bit size, we find the posit method of expressing the power-of-two scaling frees up more bits for the fraction over a wide range, giving a greater maximum accuracy for different *es* value, showing in Table 2

Table 2: Accuracy Compared with posit and float

Size, bit	Float Max Accuracy, bits	Posit Max Accuracy, bits	Range where accuracy: posit ≥ float
8	5	6	1/4 to 4
16	11	13	1/64 to 64
32	24	28	$1. \times 10^{-6}$ to $1. \times 10^{6}$

We then measure the area and power cost of posit in hardware, including float-to-posit converter, posit-to-float converter, posit adder and posit multiplier, then we compare the cost with float-point and fixed-point hardware. The logic components of the these systems are synthesized with Synopsis Design Compiler and laid out with Cadence Encounter with 90nm library. The circuit is clocked at 100MHz. Power and area measurement are reported in Table 3 and Table 4.

Table 3: Cost of float-to-posit and posit-to-float converter

Type	Power(nW)	Area(μm^2)
32-bit Float to 8-bit Posit	19841.83	774.49
8 bit Posit to 32-bit Float	19211.70	695.01

Table 4: Cost of posit, fixed-point, float-point adder and posit multiplier

Type	Power(nW)	Area(μm^2)
8-bit Posit Adder	7847.93	274.32
8 bit Posit Multiplier	298139.76	5090.91
8-bit Fixed-point Adder	8465.93	336.58
32-bit Float-point Adder	27753.62	1153.33

Table 3 shows the power and area of 32-bit Float to 8-bit Posit converter and 8-bit Posit t 32-bit Posit converter. Table 4 shows the power and area of different computational blocks. The cost of 8-bit posit adder is approximately 25% of 32-bit float-point adder. Most interestingly, we notice that the cost of the posit adder is slightly smaller than the fixed-point adder which demonstrates the benefits of posit and its broad application prospects.

6 PROTOTYPE LSTM ENGINE

We design a prototype of accelerator engine that is highly customized for LSTM computation. The equations below summarize the computations that are done for a single LSTM step. Recall that an LSTM layer consists of several time steps using the same weight matrices to process an input sequence. In the equations, there are three input vectors: x_t the input data at time t, h_{t-1} previous hidden state, and c_{t-1} the previous cell state.

$$
\begin{aligned}
i_t &= \sigma(W_{ii}x_t + W_{hi}h_{t-1}) && (3)\\
f_t &= \sigma(W_{if}x_t + W_{hf}h_{t-1}) && (4)\\
g_t &= \tanh(W_{ig}x_t + W_{hg}h_{t-1}) && (5)\\
o_t &= \sigma(W_{io}x_t + W_{ho}h_{t-1}) && (6)\\
c_t &= \sigma(f_t * c_{t-1} + i_t * g_t) && (7)\\
h_t &= \tanh(c_t) * o_t && (8)
\end{aligned}
$$

We can interpret the equations as follows:

(1) The first four equations are similar to the fully-connected layers of a deep neural network such that we do two matrix-vector multiplications for each equation and add the outputs together to get pre-activation values.
(2) Activation function *tanh* or σ (σ refers to sigmoid) is performed on the the sum of each pair of matrix-vector multiplications.
(3) The fifth line calculates the cell state output c_t from f_t, i_t, g_t and previous cell state c_{t-1}.
(4) The sixth line calculates the hidden state output h_t from c_t and o_t, which is used for the input of the next layer as well as the previous hidden state for next time step

Figure 4 is the architectural diagram of our prototype LSTM engine. Note that the attention-based model we use has input vector size of 800, which means it requires 5M Bytes cache memory to store all weight parameters if 8-bit fix-point datatype is used. As a result, only one output entry of c_t and h_t is computed at the same time such that only a single row of a each weight matrix is required to store inside the LSTM engine. The datapath of LSTM engine can mainly be divided into two parts:

(1) After x_t and h_{t-1} and a single row of each weight matrix is stored into cache buffers, we use 8 MACs to do matrix-vector multiplication and accumulate the pre-activations of f_t, i_t, g_t, o_t (only one entry) by 4 accumulators.
(2) Compute activation unit first compute four activation values and then use them to generate one entry of c_t and h_t

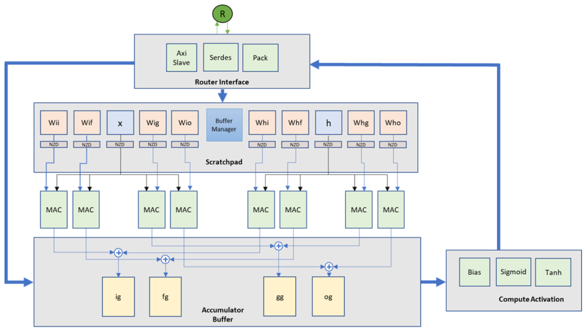


Figure 4: LSTM Accelerator Engine

The activation functions, $tanh$ and σ, are implemented in the Algorithmic C Math Library [2] using piece-wise linear approximations. Figure 5 shows that the line segments in the red curve is very close to the reference curve.

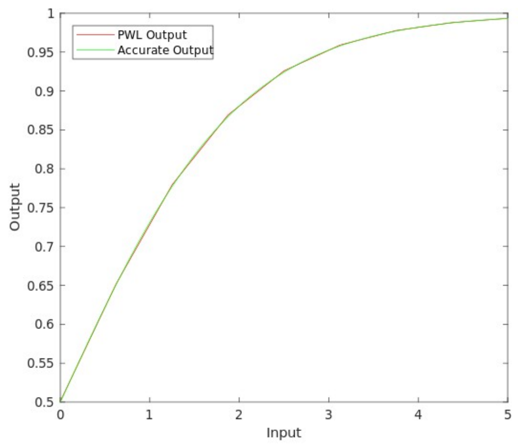


Figure 5: Sigmoid PWL Output vs. Accurate Sigmoid Output [2]

The proposed LSTM engine is simulated with random values to verify its functionality. Moreover, we have successfully synthesized it from SystemC code to RTL in Verilog with accuracy cycle behavior. We think the main bottleneck to design the whole LSTM accelerator is the efficient storage of weight matrices and a dataflow that takes advantage of weight reuse across timesteps.

7 CONCLUSION

In this project, we have shown that the posit number system is well suited for speech recognition inference at 8-bit precision. Moreover, posit-based hardware would be less expensive in terms of power and area compared to float or fixed-point. Therefore, posits present themselves as a very appealing solution for deploying compressed DNNs on the edge and warrant further study.

REFERENCES

[1] Intel unveils new roadmap. https://www.extremetech.com/computing/275102-intel-unveils-new-roadmap-new-14nm-chips-in-2019-10nm-ice-lake-in-2020, 2018.
[2] Algorithmic C Math Library. https://github.com/hlslibs/ac_math. Accessed: 2018-12-11.
[3] J. Ba, R. Kiros, and G. E. Hinton. Layer normalization. *CoRR*, abs/1607.06450, 2016.
[4] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. *CoRR*, abs/1409.0473, 2014.
[5] E. Battenberg, J. Chen, R. Child, A. Coates, Y. Gaur, Y. Li, H. Liu, S. Satheesh, D. Seetapun, A. Sriram, and Z. Zhu. Exploring neural transducers for end-to-end speech recognition. *CoRR*, abs/1707.07413, 2017.
[6] W. Chan, N. Jaitly, Q. V. Le, and O. Vinyals. Listen, attend and spell. *CoRR*, abs/1508.01211, 2015.
[7] C. Chiu, T. N. Sainath, Y. Wu, R. Prabhavalkar, P. Nguyen, Z. Chen, A. Kannan, R. J. Weiss, K. Rao, K. Gonina, N. Jaitly, B. Li, J. Chorowski, and M. Bacchiani. State-of-the-art speech recognition with sequence-to-sequence models. *CoRR*, abs/1712.01769, 2017.
[8] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan. Deep learning with limited numerical precision. *CoRR*, abs/1502.02551, 2015.
[9] Gustafson and Yonemoto. Beating floating point at its own game: Posit arithmetic. *Supercomput. Front. Innov.: Int. J.*, 4(2):71–86, June 2017.
[10] S. Han, H. Mao, and W. J. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. *CoRR*, abs/1510.00149, 2015.
[11] G. Klein, Y. Kim, Y. Deng, J. Senellart, and A. M. Rush. Opennmt: Open-source toolkit for neural machine translation. *CoRR*, abs/1701.02810, 2017.
[12] D. D. Lin, S. S. Talathi, and V. S. Annapureddy. Fixed point quantization of deep convolutional networks. *CoRR*, abs/1511.06393, 2015.
[13] M. Luong, H. Pham, and C. D. Manning. Effective approaches to attention-based neural machine translation. *CoRR*, abs/1508.04025, 2015.
[14] V. Panayotov, G. Chen, D. Povey, and S. Khudanpur. Librispeech: An asr corpus based on public domain audio books. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5206–5210, April 2015.
[15] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernandez-Lobato, G.-Y. Wei, and D. Brooks. Minerva: Enabling Low-Power, Highly-Accurate Deep Neural Network Accelerators. In *ISCA*, 2016.
[16] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, J. Klingner, A. Shah, M. Johnson, X. Liu, L. Kaiser, S. Gouws, Y. Kato, T. Kudo, H. Kazawa, K. Stevens, G. Kurian, N. Patil, W. Wang, C. Young, J. Smith, J. Riesa, A. Rudnick, O. Vinyals, G. Corrado, M. Hughes, and J. Dean. Google's neural machine translation system: Bridging the gap between human and machine translation. *CoRR*, abs/1609.08144, 2016.