DOI: 10.14529/jsfi190304

Optimizing Deep Learning RNN Topologies on Intel Architecture

Kunal Banerjee[1], *Evangelos Georganas*[2], *Dhiraj D. Kalamkar*[1], *Barukh Ziv*[3], *Eden Segal*[3], *Cristina Anderson*[4], *Alexander Heinecke*[2]

© The Authors 2019. This paper is published with open access at SuperFri.org

Recurrent neural network (RNN) models have been found to be well suited for processing temporal data. In this work, we present an optimized implementation of vanilla RNN cell and its two popular variants: LSTM and GRU for Intel Xeon architecture. Typical implementations of these RNN cells employ one or two large matrix multiplication (GEMM) calls and then apply the element-wise operations (sigmoid/tanh) onto the GEMM results. While this approach is easy to implement by exploiting vendor-optimized GEMM library calls, the data reuse relies on how GEMMs are parallelized and is sub-optimal for GEMM sizes stemming from small minibatch. Also, the element-wise operations are exposed as a bandwidth-bound kernel after the GEMM which is typically a compute-bound kernel. To address this discrepancy, we implemented a parallel blocked matrix GEMM in order to (a) achieve load balance, (b) maximize weight matrix reuse, (c) fuse the element-wise operations after partial GEMM blocks are computed and while they are hot in cache. Additionally, we bring the time step loop in our cell to further increase the weight reuse and amortize the overhead to transform the weights into blocked layout. The results show that our implementation is generally faster than Intel MKL-DNN library implementations, e.g. for RNN, forward pass is up to ∼3× faster whereas the backward/weight update pass is up to ∼5× faster. Furthermore, we investigate high-performance implementations of sigmoid and tanh activation functions that achieve various levels of accuracy. These implementations rely on minimax polynomial approximations, rational polynomials, Taylor expansions and exponential approximation techniques. Our vectorized implementations can be flexibly integrated into deep learning computations with different accuracy requirements without compromising performance; in fact, these are able to outperform vectorized and reduced accuracy vendor-optimized (Intel SVML) libraries by 1.6–2.6× while speep up over GNU libm is close to two orders of magnitude. All our experiments are conducted on Intel's latest CascadeLake architecture.

Keywords: LSTM, Intel Xeon, GEMM, compute-bound kernel, bandwidth-bound kernel.

Introduction

RNN models, unlike typical feed-forward artificial neural network models, allow connections between nodes to form a directed graph along a temporal sequence and hence, by design, are well equipped to learn from temporal data. These models have found applications in language translation [24], text generation [23], handwriting recognition [15] and image captioning [8] among many others. A particular variant of RNN called long short-term memory (LSTM) provides improvements over traditional RNN by handling exploding and vanishing gradient problems encountered during RNN training [16]. Another variant of RNN called gated recurrent unit (GRU) has been proposed which has fewer parameters than LSTM [10]. However, the choice between LSTM and GRU is not always clear and may depend on the dataset and/or the task at hand [10]. Therefore, it is important to have efficient implementations of the aforementioned RNN models in order to expedite training and inference of the various RNN based applications, especially, if these applications undergo continuous learning and/or deployed in scenarios where these are expected to perform real-time predictions.

[1]Intel Corporation, Bangalore, India
[2]Intel Corporation, Santa Clara, USA
[3]Intel Corporation, Haifa, Israel
[4]Intel Corporation, Oregon, USA

Let us now dive into the RNN cell. It is pertinent to note that non-linear activation functions, such as tanh and sigmoid (we use the term *sigmoidal* in this paper to refer to either of these two activation functions) help with the generalization of the models and the differentiation between the outputs. However, these functions are computationally expensive even in single precision since their definitions require the calculation of the exponential function. To make things even worse, emerging deep-learning hardware focuses mostly on the acceleration of the GEMM-flavored computational kernels and consequently the fraction of the time spent in such non-linear activation functions becomes even larger. Nevertheless, the current trend in deep learning is to use lower precision (e.g. float16, bfloat16 [1], int16, int8) for activations in both inference and training [12, 13] and as a result high-performance, reduced precision or lower accuracy activation functions are a viable option.

In this paper, we present implementation of a vanilla RNN cell (with support for different non-linearities) and the two popular variants, namely LSTM and GRU, which are specifically tuned for Intel Xeon architecture. In the process, we showcase benefits of our approach over Intel® Math Kernel Library for Deep Neural Networks (MKL-DNN) [2] implementation of these RNN variants, which is an open source performance library from Intel, intended for acceleration of deep learning frameworks on Intel architecture. Moreover, we focus on high-performance and reduced precision implementations of sigmoidal functions that are a natural fit for RNNs and deep learning, in general. In particular, we investigate approximation techniques based on: (a) Padé rational polynomials [7], (b) piecewise minimax polynomials [20] and (c) approximations of the exponential function via Taylor expansions [11]. Our implementations are vectorized with AVX512 instructions targeting modern Intel CPUs. We additionally explore the trade-off between accuracy and speed of these approximations. All our code is publicly available at [3] as part of the LIBXSMM library.

Rest of this paper is organized as follows. Section 1 provides an overview of the RNN cell and its variants along with the details of the steps taken to optimize these on Intel Xeon architecture. Section 2 describes the various approximation algorithms for sigmoidal functions along with their implementations. Section 3 covers the experimental results; specifically, we characterize the performance of our cell implementations vis-a-vis those of MKL-DNN; we further showcase a comparative analysis of the various approximations of sigmoidal functions on Intel Xeon architecture.

1. Implementation of RNN Cell and its Variants

1.1. (Vanilla) RNN Cell

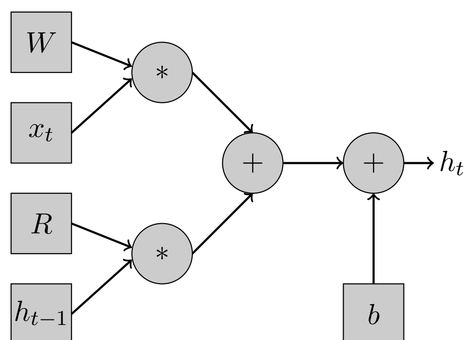


Figure 1. A diagram of an RNN cell

RNN models have found staggering success in learning from temporal data as documented in [18]. A diagram of an RNN cell is shown in Fig. 1. The equation representing one forward step of an RNN cell from timestep $t-1$ to t is given below:

$$h_t = \Lambda(W * x_t + R * h_{t-1} + b),$$

where h is the hidden state of the RNN, x is the input from the previous layer, W is the weight matrix for the input, R is the weight matrix for the recurrent connections, b is the bias and Λ is a non-linear function which can be ReLU or tanh or sigmoid (σ) (while the former two non-linearities are supported by cuDNN [9], the last one has been used by Baidu [14] – we support all three variants in our implementation; note that the backpropagation equations depend on the choice of Λ).

Typically, RNN implementations involve two GEMMs: $W * x$ and $R * h$, or one GEMM: concatenated WR with concatenated xh. The GEMM operation is followed by application of element-wise operation Λ on the GEMM result. While this approach is easy to implement by exploiting vendor-optimized GEMM library calls, the data reuse relies on how GEMMs are parallelized and is sub-optimal for GEMM sizes stemming from small minibatch. Also, the element-wise operations are exposed as a bandwidth-bound kernel after the GEMM which is typically a compute-bound kernel. To address this inconsistency, our optimization of the LSTM cell is based on a "data-flow" approach. We implemented a parallel blocked matrix GEMM in order to (a) achieve load balance, (b) maximize weight matrix reuse and (c) fuse the element-wise operations after partial GEMM blocks are computed and while they are hot in cache. Internally, we use a blocked matrix layout for weights and traditional activation format so that we exploit a better locality and avoid conflict misses. Additionally, we bring the time step LSTM loop in our cell to further increase the weight reuse and amortize the overhead to transform the weights into the blocked layout. Algorithm 1 captures the method described above succinctly.

1.2. LSTM Cell

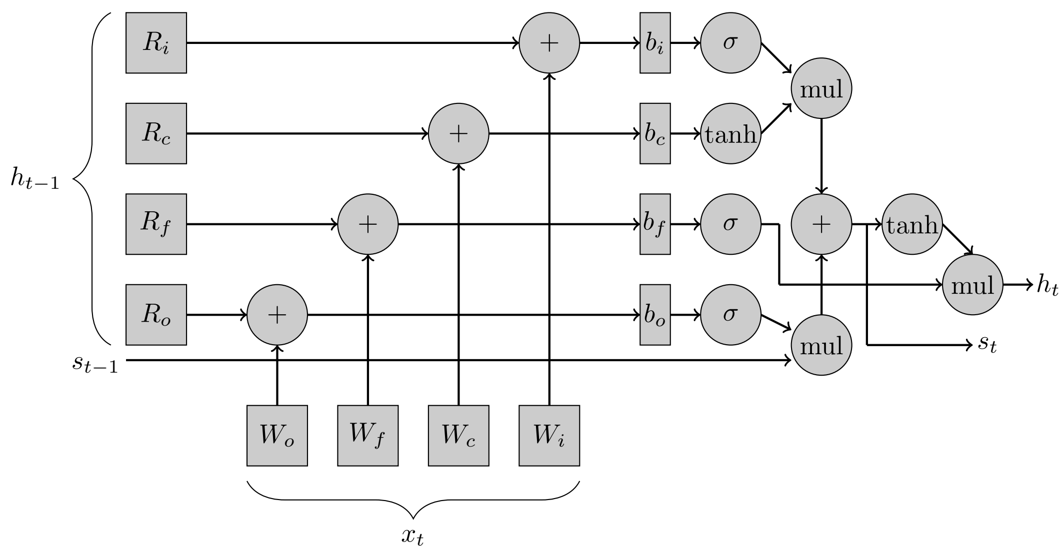


Figure 2. A diagram of an LSTM cell

A diagram of an LSTM cell is given in Fig. 2. The equations to compute the output of the LSTM cell at time step t are as follows:

Algorithm 1 Forward propagation pass of RNN cell

Inputs: N: batch size, C: input channel size, K: output channel size, T: time steps, Weight tensors W, R, bias b, input sequences x, h, blocking factors b_N, b_C, b_K

Output: Output sequence h

1: Convert W, R, x_t, h_{t-1} into blocked format. //cf. Section 3.1
2: Based on thread_id i, calculate corresponding output block of h_t: let its corner indices be N_i^{start}, N_i^{end}, K_i^{start}, K_i^{end}; let C_i^{start} and C_i^{end}, and $\mathcal{K}_i^{start}$ and $\mathcal{K}_i^{end}$ be the corresponding corner indices along input channels and output channels, correspondingly, in x_t, W and R which are required to compute h_t.
3: **for** $t = 0 \ldots T-1$ {
4:   **for** $k = K_i^{start} \ldots K_i^{end}$ {
5:     **for** $n = N_i^{start} \ldots N_i^{end}$ {
6:       **for** $c = C_i^{start} \ldots C_i^{end}$ {
7:         $\mathcal{A} \leftarrow$ blocked_GEMM($W[k][c][b_C][b_K]$, $x[t][n][c][b_N][b_C]$).
8:       }
9:       **for** $c = \mathcal{K}_i^{start} \ldots \mathcal{K}_i^{end}$ {
10:         $\mathcal{B} \leftarrow$ blocked_GEMM($R[k][c][b_K][b_K]$, $h[t][n][c][b_N][b_C]$).
11:       }
12:       $h[t+1][n][c][b_N][b_C] \leftarrow \Lambda(\mathcal{A} + \mathcal{B} + b[k])$.
13: } } }

$$
\begin{aligned}
i_t &= \sigma(W_i * x_t + R_i * h_{t-1} + b_i),\\
c_t &= \tanh(W_c * x_t + R_c * h_{t-1} + b_c),\\
f_t &= \sigma(W_f * x_t + R_f * h_{t-1} + b_f),\\
o_t &= \sigma(W_o * x_t + R_o * h_{t-1} + b_o),\\
s_t &= f_t \circ s_{t-1} + i_t \circ c_t,\\
h_t &= o_t \circ \tanh(s_t).
\end{aligned}
$$

Typical implementations of LSTM involve two large GEMMs: $\mathbf{W} * \mathbf{x}$ and $\mathbf{R} * \mathbf{h}$, where $\mathbf{W}$ and $\mathbf{R}$ are obtained by concatenating W_i, W_c, W_f, W_o and R_i, R_c, R_f, R_o respectively, or one *larger* GEMM: concatenated **WR** with concatenated **xh**, for example, original implementation of LSTM cell in TensorFlow has adopted this one larger GEMM approach. This GEMM step is followed by application of element-wise operations (sigmoid/tanh) on the GEMM results. Our implementation of LSTM cell follows the same principle as in Algorithm 1 with the exception that steps 6–11 are repeated four times (one each for i, c, f and o), and instead of single step 12, there are multiple element-wise operations which are applied to compute s and h.

1.3. GRU Cell

It is important to note that we have found subtle differences in the equations for GRU across different implementations, e.g., those between cuDNN [9] and TensorFlow [5]. We have adopted the formulation provided in TensorFlow (to aid in future integration) as stated below:

$$
\begin{aligned}
i_t &= \sigma(W_i * x_t + R_i * h_{t-1} + b_i),\\
c_t &= \sigma(W_c * x_t + R_c * h_{t-1} + b_c),\\
o_t &= h_{t-1} \circ i_t,\\
f_t &= \tanh(W_f * x_t + R_f * o_t + b_f),
\end{aligned}
$$

$$h_t = (1 - c_t) \circ f_t + c_t \circ h_{t-1}.$$

We have followed the same design policies as mentioned for LSTM and hence we refrain from further elaboration for brevity.

2. Implementations for Approximate Sigmoidal Functions

For neural networks, activation functions comprise an important feature that essentially determines whether neurons in the network should be activated or not. The most widely used activation functions are nonlinear like tanh and sigmoid because such functions help with the generalization of the models and the differentiation between the outputs. However, these functions are computationally expensive because their definitions require the calculation of the exponential function. Therefore, it is worth investigating algorithms which can approximate these non-linear activation functions without sacrificing accuracy considerably. In this work, we focus on algorithms which can be easily implemented in software with Intel's existing 512bit SIMD instruction set, and consequently, do not require any specialized hardware, such as the ones reported in [19, 22].

For the remainder of this paper, we will be considering only the tanh function given that the standard logistic sigmoid is a rescaled tanh:

$$sigmoid(x) = (\tanh(x/2) + 1)/2.$$

2.1. Rational Padé Approximations

The tanh function has two asymptotes, therefore approximating it with a mere Taylor expansion of low degree would result in poor approximation. Instead we consider rational approximations of tanh and more specifically the Padé rational polynomials. The rational Padé approximation $Padé[p/q]f(x)$ of a function f is a ratio of two polynomials with degrees p and q:

$$Padé_{[p/q]f}(x) = \frac{\sum_{i=0}^{p} a_i x^i}{\sum_{i=0}^{q} b_i x^i}.$$

which agrees with f to the highest possible order, in essence:

$$\begin{aligned} f(0) =& Padé_{[p/q]f}(0), \\ f'(0) =& Padé'_{[p/q]f}(0), \\ & \vdots \\ f^{(p+q)}(0) =& Padé^{(p+q)}_{[p/q]f}(0). \end{aligned}$$

To calculate the coefficients a_i and b_i one can consider the first $p + q$ derivatives of f at zero and finally solve the corresponding system of equations. By construction, the error of $Padé[p/q]f(x)$ agrees with the truncation error of the Taylor series about 0 truncated at the $(p + q)^{th}$ term.

The implementation of the $Padé_{[3/2]}(x)$ for tanh with AVX512 instructions is shown in Fig. 3 (for simplicity we do not include the initialization of the constant vectors). This implementation uses Fused Multiply and Accumulate (FMA) instructions to evaluate the polynomials in the numerator and the denominator of the rational approximation via the Horner's rule. For performance reasons, instead of dividing the numerator by the denominator, we use the approximate

reciprocal intrinsic (line 5 in Fig. 3) and a multiplication. Also, in order to ensure that the function is bound within the asymptotes at $y = \pm 1$, we clip it for input values beyond two limits lo and hi by leveraging two compare and two blend instructions (lines 7–10). In Section 3.2, we also include the evaluation of the $Padé_{[7/8]}(x)$ approximation which performs more FMAs for the respective polynomial evaluations but also achieves higher accuracy.

```
__m512 x2        = _mm512_mul_ps(x, x);                  //line  1
__m512 t1_nom    = _mm512_fmadd_ps(x2, c1, one);         //line  2
__m512 nom       = _mm512_mul_ps(t1_nom, x);             //line  3
__m512 denom     = _mm512_fmadd_ps(x2, c2, one);         //line  4
__m512 denom_rcp = _mm512_rcp14_ps(denom);               //line  5
__m512 result    = _mm512_mul_ps(nom,denom_rcp);         //line  6
__mmask16 maskHi = _mm512_cmp_ps_mask(x, hi, _CMP_GT_OQ); //line  7
__mmask16 maskLo = _mm512_cmp_ps_mask(x, lo, _CMP_LT_OQ); //line  8
result = _mm512_mask_blend_ps(maskHi, result, one);      //line  9
result = _mm512_mask_blend_ps(maskLo, result, neg_one);   //line 10
```

Figure 3. Rational Padé 3/2 approximation

2.2. Piecewise Minimax Polynomials Approximations

Another approach for approximating transcendental functions is to use piecewise polynomial approximations. Various polynomials may be chosen for approximation; in this section we leveraged minimax polynomials [20]. Therefore, we divide the input range of the function $\tanh(x)$ into intervals and for each interval $[a, b]$ we find a polynomial p of degree at most n to minimize:

$$\max_{a \leq x \leq b} |\tanh(x) - p(x)|.$$

In our approximations, we utilize truncated Chebyshev series [21] that closely approximate the minimax polynomials.

Figure 4 shows the implementation of the tanh function approximation using minimax polynomials of second degree. First, the function's input range is divided into 16 intervals using the argument's exponent and the Most Significant Bit (MSB) and an index register `idx` is generated (lines 2–5). Then, the code uses 3 lookup tables (`tanh_c0_reg`, `tanh_c1_reg` and `tanh_c2_reg`) consisting of 16 entries each to simultaneously load 16 triples of coefficients of the approximating polynomial. In order to load the coefficients in three registers, we utilize the `_mm512_permutexvar_ps` intrinsic which shuffles single-precision values in a zmm register using the corresponding index `idx` (lines 6–8). This in-register look-up table is much faster ($\sim$4$\times$) than using AVX512's gather instructions from memory. The polynomial evaluation is materialized with FMAs and the Horner's rule. Only the positive range of inputs is represented; maenwhile, for negative input values, we exploit the property $\tanh(-x) = -\tanh(x)$ resulting from the point symmetry (line 11).

There are two ways to reduce the approximation error:

Divide the input range to more intervals: Increasing the number of intervals from 16 to 32 will require 3 additional zmm registers to hold the coefficients. The number of instructions will not change but the loading of the coefficients will have a greater latency because we have to use the `_mm512_permutex2var_ps` instruction which shuffles single-precision elements in two zmm

registers using the corresponding selector/index in `idx`.

Use polynomials of higher degree: Every additional degree will need an additional zmm register to store the coefficients, one additional instruction to load the coefficients and one additional FMA to evaluate the polynomial. In Section 3.2, we also evaluate the tanh approximation using minimax polynomials of third degree.

```
__m512  signs   = _mm512_and_ps(x, ps_sign_mask);                  //line  1
__m512  abs_x   = _mm512_and_ps(x, ps_sign_filter);                //line  2
__m512i idx     = _mm512_srli_epi32(_mm512_castps_si512(abs_x),22); //line  3
idx             = _mm512_max_epi32(idx, lut_low);                  //line  4
idx             = _mm512_min_epi32(idx, lut_high);                 //line  5
__m512 c0       = _mm512_permutexvar_ps(idx, tanh_c0_reg);         //line  6
__m512 c1       = _mm512_permutexvar_ps(idx, tanh_c1_reg);         //line  7
__m512 c2       = _mm512_permutexvar_ps(idx, tanh_c2_reg);         //line  8
__m512 result   = _mm512_fmadd_ps(abs_x, c2, c1);                  //line  9
result          = _mm512_fmadd_ps(abs_x, result, c0);              //line 10
result          = _mm512_xor_ps(result, signs);                    //line 11
```

Figure 4. Approx. with 2nd degree minimax polynomials

2.3. Tanh via exp() Approximation with Taylor Series

In this approximation method (see Fig. 5), we use the definition of tanh:

$$\tanh(x) = 1 - \frac{2}{1 + e^{2x}}$$

and we approximate the calculation of e^{2x}.

In order to approximate the function e^x, we exploit the property $e^x = 2^{x \log_2 e} = 2^{n+y} = 2^n \cdot 2^y$ with $n = round(x \log_2 e)$ and $y = x \log_2 e - n$. With these properties in mind, all we have to do is to compute the term 2^n with n being an integer, and the term 2^y with $|y| \in [0, 1)$. For the term 2^y we use a second-degree Taylor polynomial (lines 3-4). Once 2^y is calculated, we leverage the instruction `_mm512_scalef_ps(A,B)` which returns a zmm holding $a_i \cdot 2^{floor(b_i)}$ for each $a_i \in A$ and $b_i \in B$. This scale instruction (line 5) concludes the approximation of the exponential part in the denominator. The approximation error can be further reduced by using a larger-degree Taylor expansion for the 2^y term; in Section 3.2, we also evaluate a third-degree Taylor polynomial.

```
__m512 _x       = _mm512_fmadd_ps(x, twice_log2_e, half); //line 1
__m512 y        = _mm512_reduce_ps(_x, 1);                //line 2
__m512 t1       = _mm512_fmadd_ps(y, c2, c1);             //line 3
__m512 two_to_y = _mm512_fmadd_ps(y, t1, c0);             //line 4
__m512 exp      = _mm512_scalef_ps(two_to_y, _x);         //line 5
__m512 den_rcp = _mm512_rcp14_ps(_mm512_add_ps(exp,one)); //line 6
__m512 result  = _mm512_fmadd_ps(den_rcp,minus_two,one);  //line 7
```

Figure 5. Tanh via exp() approximation

3. Experimental Results

3.1. Evaluation of RNN Cell and its Variants

Intel® Math Kernel Library for Deep Neural Networks (MKL-DNN) is an open source performance library from Intel intended for acceleration of deep learning frameworks on Intel architecture. Hence, we choose to compare our LSTM cell with that of MKL-DNN (version 1.0.2). We also compare it with BLAS where we perform $\mathbf{W} * \mathbf{x}$ and $\mathbf{R} * \mathbf{h}$ followed by element-wise operations for the forward pass; note that we cannot concatenate weights and perform a single large BLAS call (with enhanced performance) in the backward/weight update pass and hence we omit comparison with BLAS during this pass. Note that all the numbers reported in this subsection are measured on Intel® Xeon® Platinum 8280 processor codenamed **CascadeLake (CLX)** with 28 cores at 2.4 GHz AVX turbo frequency.

3.1.1. RNN cell efficiency

As mentioned earlier, we have adopted a "dataflow"-based approach for optimizations. We use blocked layout to better exploit locality and avoid conflict misses. Given N = minibatch size, C = input channels and K = output channels and T = total time steps, internally, we transform the inputs in blocked format as mentioned below:

- input activations: $[T][N][C] \rightarrow [T][N/B_N][C/B_C][B_N][B_C]$;
- hidden activations: $[T][N][K] \rightarrow [T][N/B_N][K/B_K][B_N][B_K]$;
- weights: $[C][K] \rightarrow [K/B_K][C/B_C][B_C][B_K]$;
- recurrent weights: $[K][K] \rightarrow [K/B_K][K/B_K][B_K][B_K]$,

where B_N, B_C and B_K are blocking factors for N, C and K, respectively. We perform computation with fused-time steps which amortizes the cost of blocking.

The heart of our blocked matrix GEMM consists of a JIT-ed small batch-reduce GEMM kernel which we implemented in LIBXSMM. The batch-reduce GEMM materializes the operation

$$C = \sum_{i=1}^{n} A_i * B_i$$

while it keeps the C accumulator in registers. Once a block of GEMM is computed, we apply element-wise operations on it while hot in cache. It is worth noting that we use Intel AVX512 intrinsics for vectorization and Intel Short Vector Math Library (SVML) for fast tanh and sigmoid computations. Same optimization principles are applied to backward and weight-update passes as well. Furthermore, our RNN operators are thread-library agnostic (can use any of pthreads, OpenMP, C++ threads, Cilk, TBB, etc.) – thus enabling an easy integration with any choice of framework.

Our experiments show that LIBXSMM RNN cell-forward pass can outperform that of MKL-DNN by $\sim 3\times$ for smaller hidden-state sizes. However, this performance's speed-up gradually decreases for larger hidden state sizes because GEMM has cubic complexity while the element-wise operations are quadratic and, as such for large sizes, the element-wise operations' bandwidth overheads are less emphasized. The results are shown in Fig. 6. Note that we have reported the floor values of all numbers for better readability. The performance difference is even more highlighted for combined backward and weight update passes as given in Fig. 7. As can be seen in this figure, LIBXSMM RNN cell outperforms that of MKL-DNN for smaller hidden states by

up to ∼5×. It is worth noting that the activation function used in Fig. 6 and Fig. 7 is tanh; we see similar patterns for sigmoid and ReLU activation functions as well.

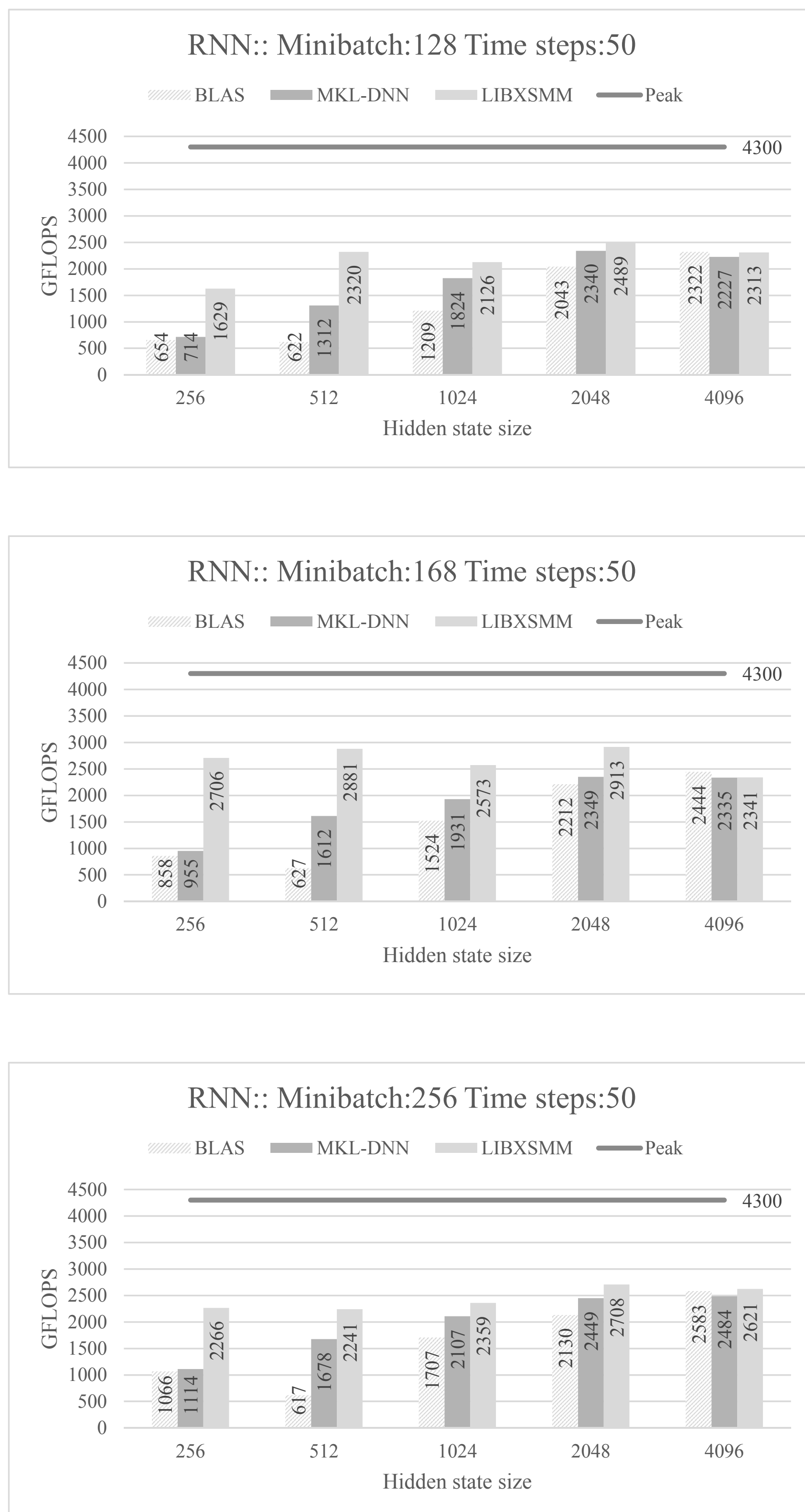


Figure 6. RNN cell-forward pass results for batch sizes: (top) 128, (middle) 168, (bottom) 256

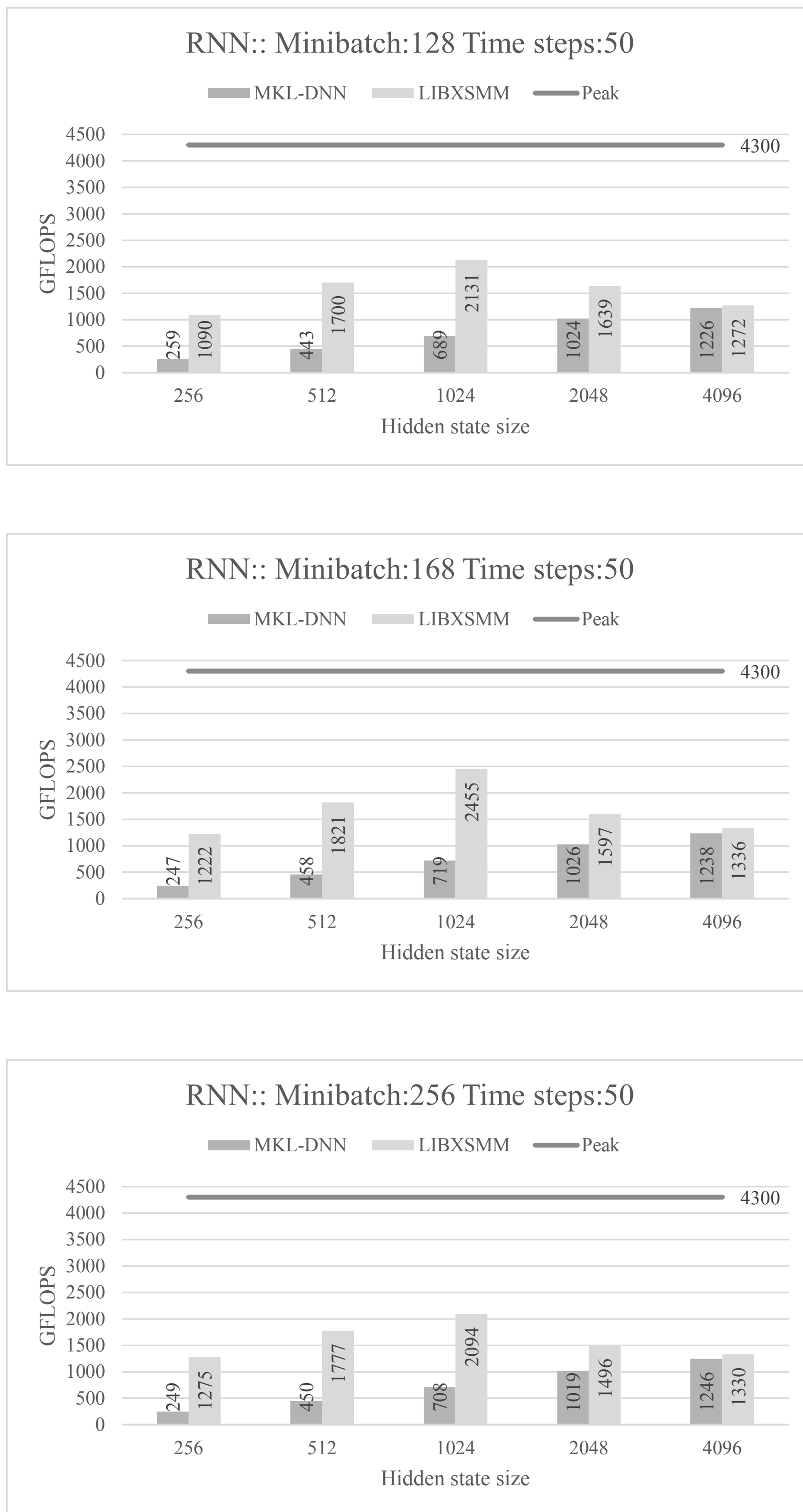


Figure 7. RNN cell-backward/weight update pass results for batch sizes: (top) 128, (middle) 168, (bottom) 256

3.1.2. LSTM cell efficiency

In LSTM, there are four different weights and recurrent weights; consequently, we block these in the following way:

- weights: $[C][4K] \rightarrow [4K/B_K][C/B_C][B_C][B_K]$;
- recurrent weights: $[K][4K] \rightarrow [4K/B_K][K/B_K][B_K][B_K]$.

We compared our optimized LSTM cell with the latest MKL-DNN LSTM cell on a single socket Xeon Platinum 8280. The LIBXSMM LSTM cell-forward pass based on our approach always outperformed that of MKL-DNN and BLAS as shown in Fig. 10. For small/medium-size problems our implementation is $\sim$1.3$\times$ faster than MKL-DNN; however, with an increase in hidden-state size, the performance of MKL-DNN becomes comparable with ours. It is worth noting that since the GEMM sizes of LSTM are four times larger than RNN for given N, C and K values, the blocking factors are accordingly smaller for LSTM, so that identical number of elements (as RNN) can fit into the cache. For the fused backward/weight-update pass, the LIBXSMM LSTM cell's performance is $\sim$2.2$\times$, on average, with respect to MKL-DNN LSTM cell as shown in Fig. 11.

3.1.3. GRU cell efficiency

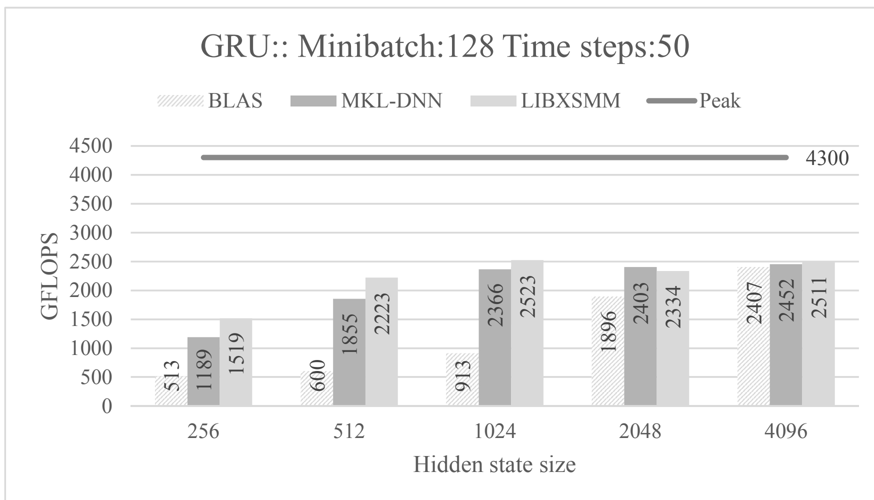
For GRU cell, which has three different weights and recurrent weights (in contrast to four of LSTM), we follow an identical policy of blocking the inputs and weights as that of LSTM. Hence, we do not elaborate on this cell. The results of its forward and backward/weight-update passes can be found in Fig. 8 and Fig. 9, respectively, where we can see that while the forward pass can be up to $\sim$1.5$\times$ faster, backward/weight-update pass can even surpass 2$\times$ speedup in comparison to MKL-DNN.

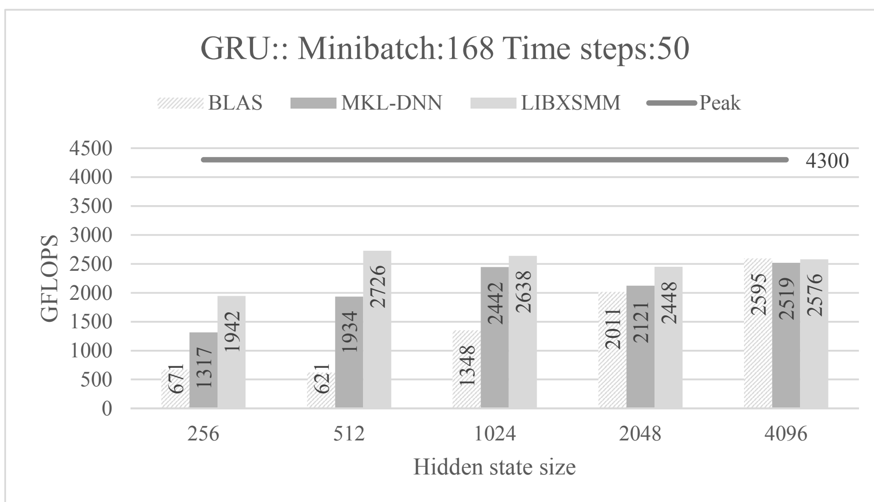
We belive a dataflow approach is better suited for CPUs than GPUs since it relies on coarse-grained parallelization and precise locality control. It is noteworthy that in LIBXSMM, the same optimizations as mentioned here have also been applied to the implementation of fully connected layer, which is the main computational kernel in multi-layer perceptron networks.

3.1.4. Application level impact of LSTM cell

Googles neural machine translation (GNMT) [24] is state-of-the-art LSTM-based language translation application. As shown in Fig. 12, we compare between four variants of GNMT code to highlight the speed-up that our code achieves for 8-layer German-to-English GNMT model. For all the experiments, we consider a minibatch size of 168 with the number of inter_op_threads equal to 1 and a number of intra_op_threads equal to 28.

1. **Reference w/o MKL:** This is the default TensorFlow code which does not support MKL.
2. **Reference w/ MKL:** This is TensorFlow with MKL support.
3. **XsmmLSTM:** This is the GNMT code which has LIBXSMM LSTM cell integrated without the support for fused time steps. This is an intermediate code that we tried out because it allowed an easy integration with other TensorFlow wrappers, thereby enabling fast deployment. Moreover, we maintained the activation and the weight layouts identical to that of TensorFlow which aided in correctness checks.
4. **+Fused Encoder:** This code supports LIBXSMM LSTM cell having fused time steps along with optimal blockings for activations and weights. However, note that the LIBXSMM LSTM cells are deployed only for the encoders. We could not change decoders to use our cell because the time step loop for decoding stage is implemented inside seq2seq library [4] and we presently leave it as a future work. Note that at this step, we achieve 2.35$\times$ performance compared to the default implementation.





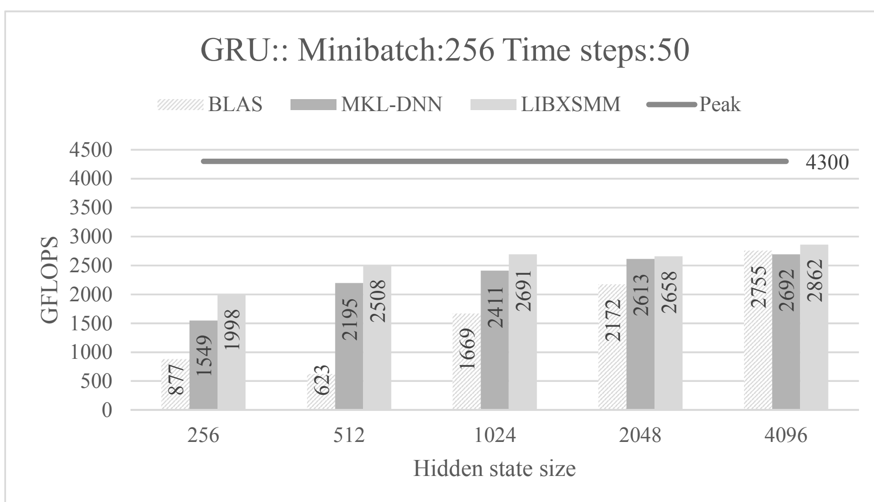
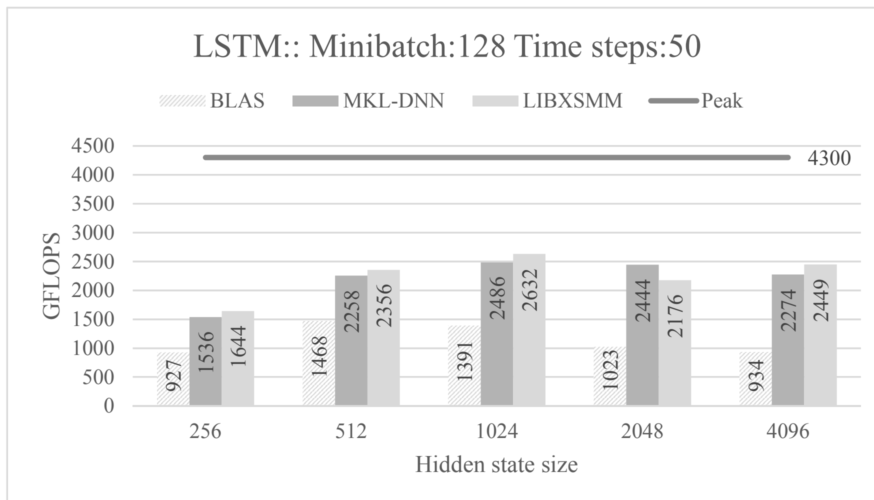
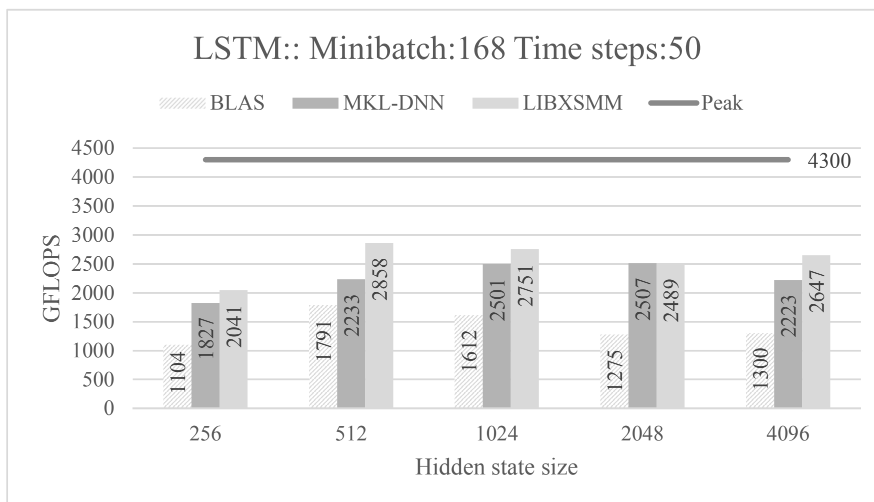


Figure 8. GRU cell forward pass results for batch sizes: (top) 128, (middle) 168, (bottom) 256

Figure 9. GRU cell backward/weight update pass results for batch sizes: (top) 128, (middle) 168, (bottom) 256





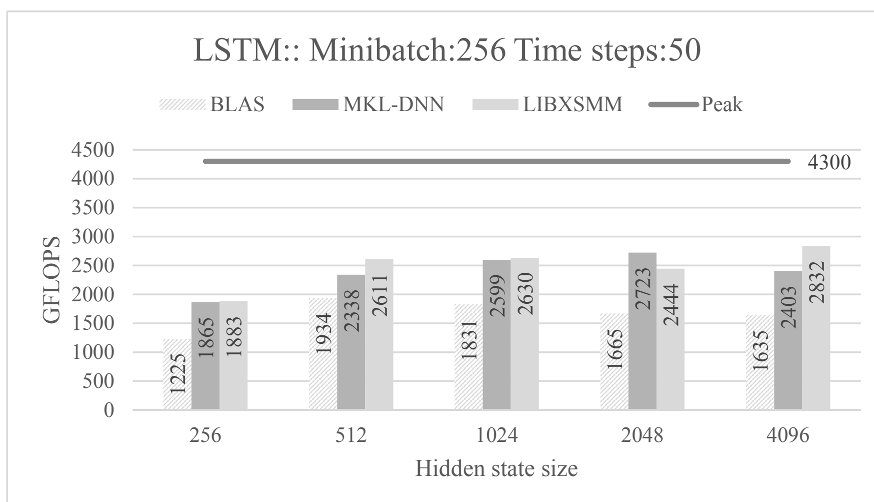


Figure 10. LSTM cell forward pass results for batch sizes: (top) 128, (middle) 168, (bottom) 256

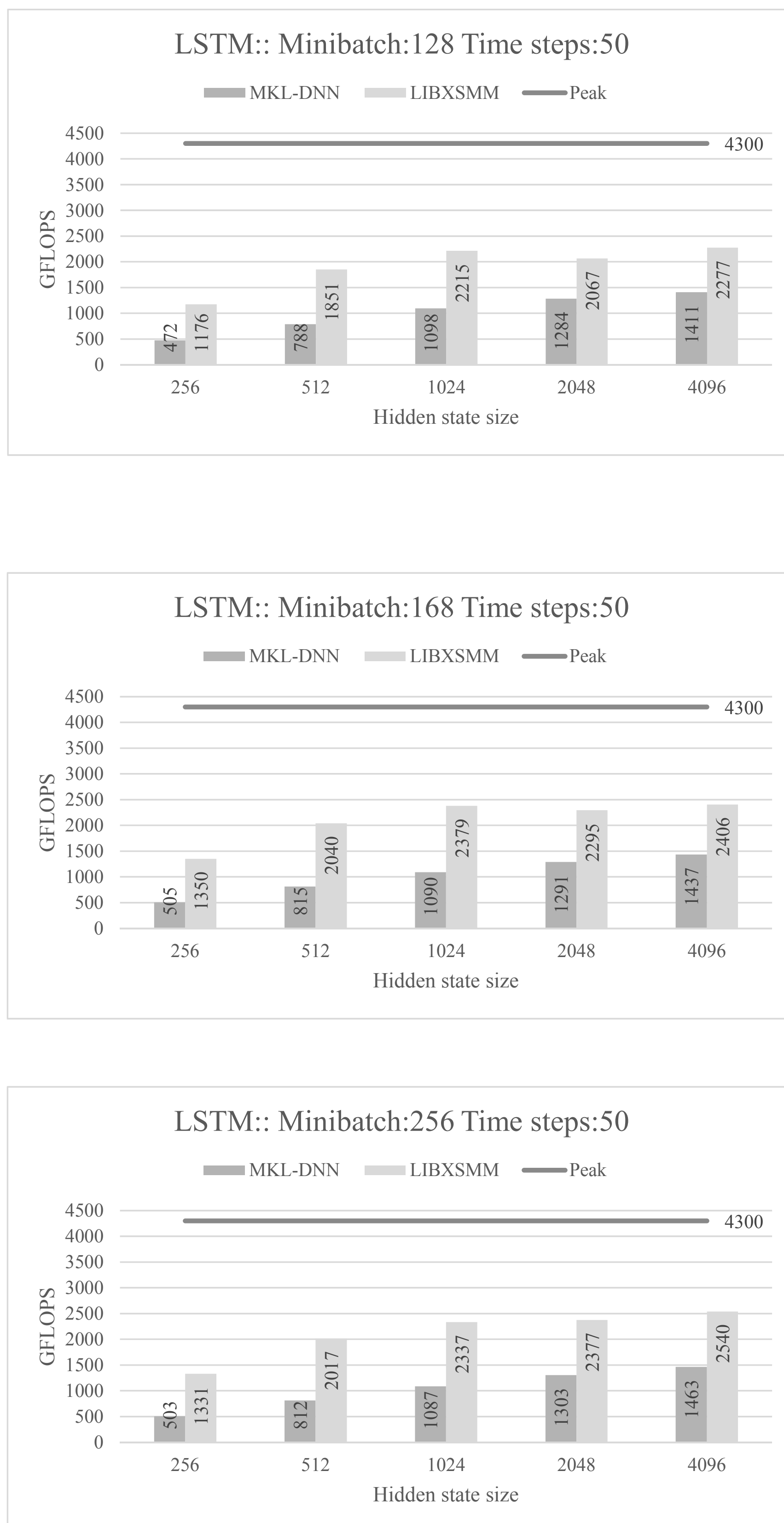


Figure 11. LSTM cell backward/weight update pass results for batch sizes: (top) 128, (middle) 168, (bottom) 256

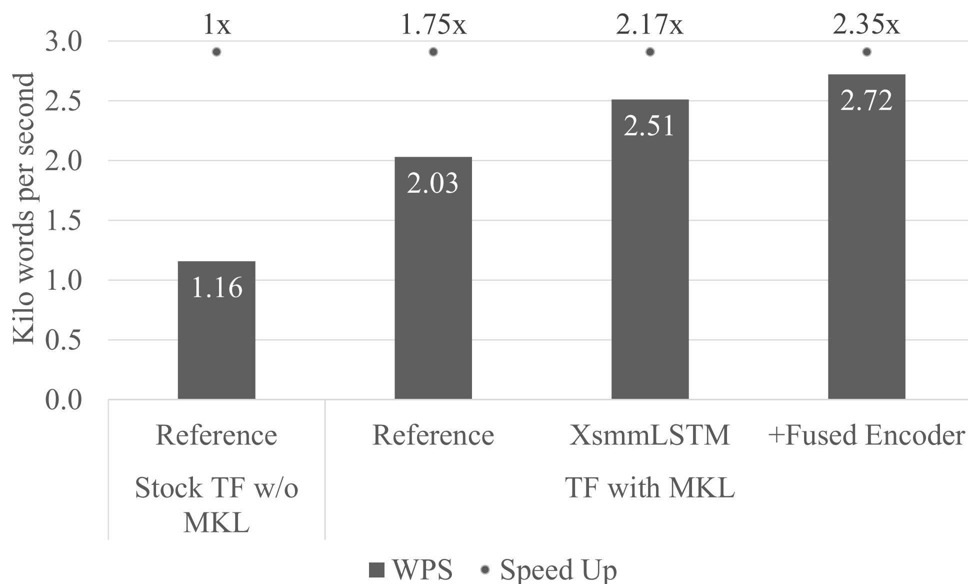


Figure 12. GNMT 4-layer performance on CascadeLake (with turbo enabled)

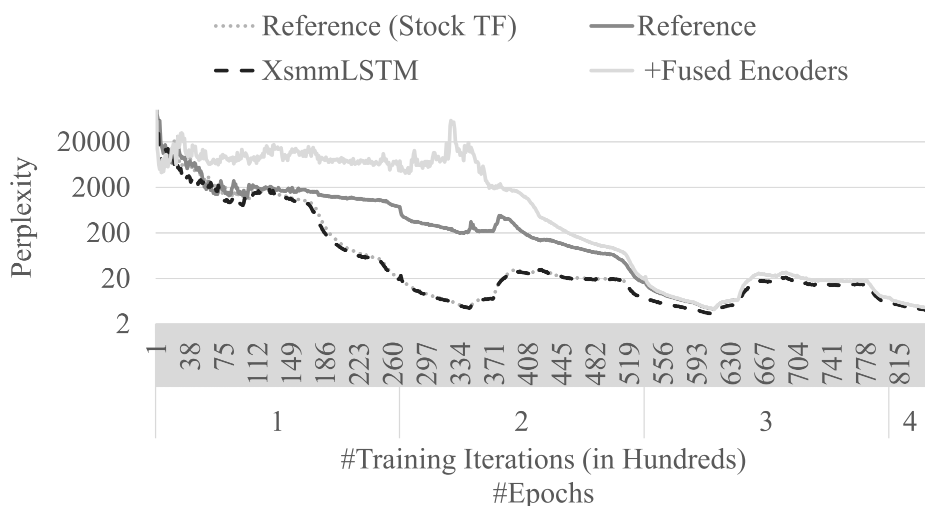


Figure 13. GNMT convergence: Perplexity

Perplexity is a measure of how easy a probability distribution is to predict; thus, the lower the value of perplexity, the better it is. Figure 13 shows how the perplexity varies during the training of the 8-layer GNMT model for the four different GNMT codes described above. As is evident from the figure, all the codes converge and follow a similar trend. In our recent work [17], we explain in detail how we have integrated our LIBXSMM LSTM cell (with fused time steps) into TensorFlow [5] framework and for the first time scaled GNMT to a 16-node Intel CPU cluster. Our code outperformed Googles stock CPU-based GNMT implementation by more than 25× on the 16 node CPU cluster. Along with the efficient scaling libraries and a smart batching strategy, the LIBXSMM LSTM cell played a crucial role in obtaining this milestone.

3.2. Evaluation of Approximate Sigmoidal Functions

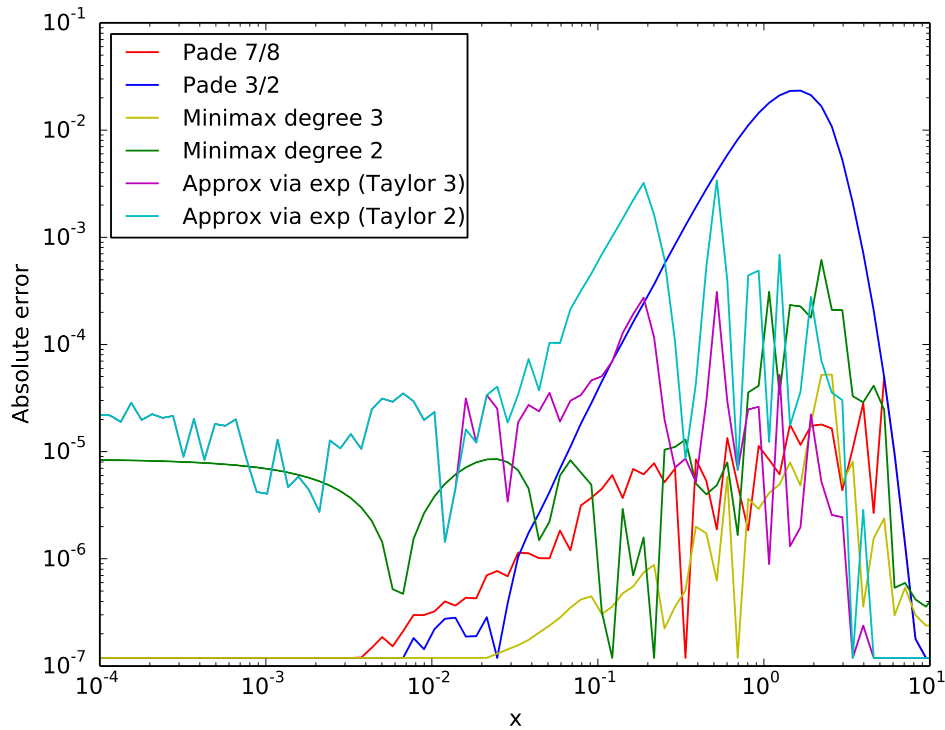


Figure 14. Absolute error of the tanh implementations on CascadeLake

In order to assess the accuracy and the performance of our approximations, we conducted experiments on Intel's CascadeLake (CLX) as described earlier.

3.2.1. Accuracy of approximations

Figure 14 (Left) shows the absolute error of the approximations on CLX in the positive range $[10^{-4}, 10]$ since tanh is symmetric with respect to the origin. In regard to the rational approximations, the $Padé_{[7/8]}$ has maximum absolute error of 10^{-4}, whereas the $Padé_{[3/2]}$ has maximum absolute error of 10^{-2}. The 2nd- and 3rd-degree piecewise minimax polynomials have maximum absolute errors of 10^{-3} and 10^{-4}, respectively. Notably, the 3rd-degree minimax polynomials exhibit a smaller absolute error than the Padé rational approximations in most of this regime. Finally, regarding the tanh via exp() approximation with Taylor polynomials of order 2 and 3, we get maximum absolute errors of 10^{-3} and 10^{-4} respectively. We also observe that the latter approximations are more accurate among all other considered variants for large x values. These observations imply that one can get an even better approximation by combining properly the aforementioned algorithms in the relevant intervals, similarly to previous work [6]. In this work, since we prioritize the speed of the approximations, we do not consider such hybrid algorithms.

At first sight these errors are undoubtedly much higher than single precision's machine epsilon of roughly 10^{-7}. However, one has to keep in mind that the emerging datatypes for deep learning training are float16 which offers a bit more than three digits precision and bfloat16 which has a bit less than three digits of precision. Deep learning inference tasks are often able to run fine with 8 bit fix-point datatypes. Given such input data, the obtained accuracies are perfectly sufficient as they are in the regime of these datatypes' machine epsilons. Figure 14 shows the absolute error of the approximations on CLX.

Table 1. Performance of the various implementations on CascadeLake

Algorithm	Cycles per tanh computation
Rational Padé 3/2	0.39
Rational Padé 7/8	0.59
Minimax polynomials of degree 2	0.35
Minimax polynomials of degree 3	0.42
Approximation via exp (Taylor degree 2)	0.42
Approximation via exp (Taylor degree 3)	0.47
SVML (high precision)	1.53
SVML (low precision)	0.95
libm	28.32

3.2.2. Performance evaluation

Table 1 shows the performance of various approximation algorithms on CLX, where the figure of merit is cycles-per-tanh computation (since all implementations leverage AVX512 instructions, each function call performs 16 tanh computations at once). In addition to our implementations, we measured the performance of the Intel Short Vector Math Library (SVML) and the tanh implementation in libm.

On CLX, the fastest implementations are the 2nd-degree minimax polynomials and the $Padé_{[3/2]}$ approximation with 0.35 and 0.39 cycles-per-tanh computation, respectively. Among these two implementations and given the absolute errors depicted in Fig. 14, we conclude that the 2nd-degree minimax polynomials yield the best tradeoff in speed and accuracy. This approximation is 2.7× faster than the low-precision SVML tanh implementation and 81× faster than the implementation in libm. The accuracy level of the 2nd-degree minimax polynomials should be sufficient for most of the low-precision deep learning applications. However, if accuracy higher then the $Padé_{[7/8]}$ requested, the 3rd-degree minimax polynomials and the tanh via exp() approximation with Taylor polynomial of order 3 may be used, yielding speedups of 1.6×, 2.3× and 2× over the low-precision SVML tanh implementation.

We further used the Intel Architecture Code Analyzer (version v3.0-28) to assess how the instructions are scheduled on CLX. As an illustrative example, we will use the assembly code of the benchmarking loop with the 3rd-degree minimax polynomials exhibited at Tab. 2. Based on the scheduling of the instructions, we observe that the critical path is determined by ports P0 and P5 where the permute, FMAs and the remaining vector compute instructions are scheduled. This critical path with the length of 6.5 cycles determines the reciprocal throughput, and this calculation agrees with out empirical result: each iteration computes 16 tanh evaluations; therefore, each tanh evaluation is estimated to take $6.5/16 = 0.41$ cycles, which is close to the measured throughput 0.47.

Finally, by considering the instruction mix in the approximations, we gain some intuition about how to create hardware that accelerates such implementations. For example, a common method used in all our approximation algorithms is the polynomial evaluation via FMA instructions that implement the Horner's rule. One could accelerate such method via fixed function hardware, e.g. by chaining k FMA units to perform a $k-$th degree polynomial evaluation. Such specialized hardware can provide faster and higher accuracy approximations by making high degree polynomial computations inexpensive.

Table 2. Assembly code and CLX instruction scheduling of the benchmarking loop with the 3rd degree minimax polynomials

Uops	Ports pressure in cycles								Instructions
	P0	**P1**	**P2**	**P3**	**P4**	**P5**	**P6**	**P7**	
1							1.0		inc eax
1			1.0						vmovups zmm10, zmmword ptr [rsp+0x600]
1	0.5					0.5			vandps zmm0, zmm10, zmm8
1	0.5					0.5			vandps zmm1, zmm10, zmm9
1	1.0								vpsrld zmm11, zmm0, 0x16
1	1.0								vpmaxsd zmm12, zmm11, zmm7
1	1.0								vpminsd zmm13, zmm12, zmm6
1						1.0			vpermps zmm15, zmm13, zmm3
1						1.0			vpermps zmm14, zmm13, zmm2
1						1.0			vpermps zmm10, zmm13, zmm4
1						1.0			vpermps zmm11, zmm13, zmm5
1	1.0								vfmadd231ps zmm15, zmm14, zmm0
1	0.5					0.5			vfmadd231ps zmm10, zmm15, zmm0
1	0.5					0.5			vfmadd213ps zmm0, zmm10, zmm11
1	0.5					0.5			vxorps zmm0, zmm0, zmm1
2				1.0	1.0				vmovups zmmword ptr [rsp+0x640], zmm0
1									cmp eax, r13d
0									jl 0xffffffffffffff94

3.2.3. Application level impact of approximate tanh

In order to study the impact of approximate tanh at the application level, we implemented LSTM cell with various approximate tanh implementations and used these LSTM cells for Google Neural Machine Translation (GNMT) training. We ran our convergence experiments on a small 4-node CLX cluster with global minibatch size of 1024 for German-to-English WMT16 training dataset using Adam solver with learning rate of 0.0005 for 5 epochs. We evaluated translation accuracy as a BLEU score after every epoch using newstest2013 as development (DEV) and newstest2015 as test (TEST) datasets. As seen in the Fig. 15, there is little impact of tanh approximation on training convergence and even though there is some variation at the beginning of the training, at the end of 5th epoch, all the evaluated versions converge to similar accuracy. Particularly, Rational Padé approximation produces best TEST BLEU score of 26.9 which is slightly better than the BLEU score of 26.7 for the reference SVML version. Similarly, a minimax polynomial-degree 3 version produces the best DEV score of 26.2 compared to 26.1 for the reference SVML version.

Conclusion

In summary, we propose an implementation of LSTM cell using a “dataflow”-approach small-blocked GEMMs instead of large GEMMs on Intel Xeon architecture. Out strategy helps in maximizing locality, weight reuse and fuse element-wise operations which are, otherwise, exposed as bandwidth-bound kernels. For small/medium sized problems, our implementation of RNN-forward pass is ∼3× faster than the MKL-DNN cell, while for backward/weight update it is up to ∼5× faster. For large weight matrices, the two approaches, however, have similar performance which stems from the fact that GEMM has cubic complexity, whereas fusing element-wise

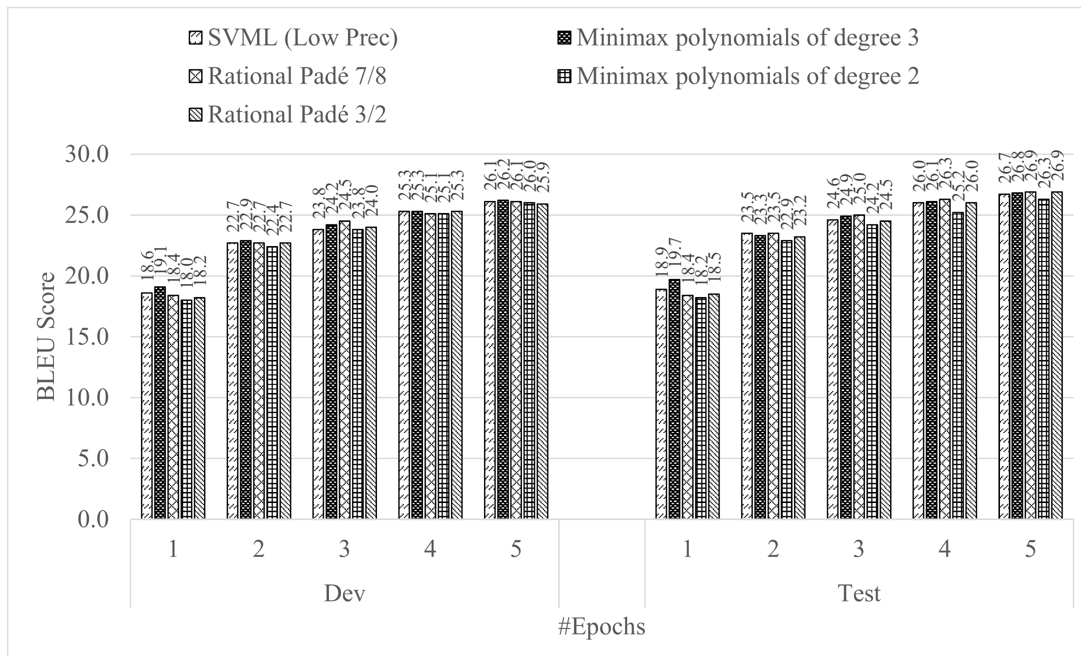


Figure 15. Convergence of GNMT with various approximate tanh implementations

operations are only of quadratic complexity. It should be noted that this conclusion might change with GEMM acceleration hardware.

We also assessed the accuracy/speed tradeoffs of the various implementations of sigmoid and tanh functions on Intel's CascadeLake processor. Our approximations obtain accuracies that are sufficient for contemporary inference and training deep learning applications with low precision (e.g. float16, bfloat16, 8 bit fix-point datatypes) while they outperform the low-precision tanh SVML implementations by factors of 1.6–2.6×. We envision that our implementations will have a magnified importance in the context of emerging deep-learning hardware that accelerates GEMM-flavored computations and, as a result, a fast evaluation of non-linear activation functions is necessitated.

This paper is distributed under the terms of the Creative Commons Attribution-Non Commercial 3.0 License which permits non-commercial use, reproduction and distribution of the work without further permission provided the original work is properly cited.

References

1. BFLOAT16 - hardware numerics definition. `https://software.intel.com/sites/default/files/managed/40/8b/bf16-hardware-numerics-definition-white-paper.pdf`, accessed: 2019-03-22

2. Intel(R) Math Kernel Library for Deep Neural Networks. `https://github.com/intel/mkl-dnn`, accessed: 2019-03-22

3. LIBXSMM. `https://github.com/hfp/libxsmm`, accessed: 2019-09-13

4. Module: tf.contrib.seq2seq. `https://www.tensorflow.org/api_docs/python/tf/contrib/seq2seq`, accessed: 2019-04-08

5. Abadi, M., Barham, P., Chen, J., *et al.*: Tensorflow: A system for large-scale machine learning. In: OSDI. pp. 265–283 (2016)

6. Beebe, N.H.: Accurate hyperbolic tangent computation. Technical report, Center for Scientific Computing, Department of Mathematics, University of Utah (1991)

7. Brezinski, C.: Outlines of padé approximation. In: Computational aspects of complex analysis, pp. 1–50. Springer (1983)

8. Chen, M., Ding, G., Zhao, S., *et al.*: Reference based LSTM for image captioning. In: AAAI. pp. 3981–3987 (2017)

9. Chetlur, S., Woolley, C., Vandermersch, P., *et al.*: cuDNN: Efficient primitives for deep learning. CoRR abs/1410.0759 (2014), `http://arxiv.org/abs/1410.0759`

10. Chung, J., Gülçehre, Ç., Cho, K., Bengio, Y.: Empirical evaluation of gated recurrent neural networks on sequence modeling. CoRR abs/1412.3555 (2014), `http://arxiv.org/abs/1412.3555`

11. Cody, W.J.: Software Manual for the Elementary Functions (Prentice-Hall series in computational mathematics). Prentice-Hall, Inc. (1980)

12. Courbariaux, M., Bengio, Y., David, J.P.: Training deep neural networks with low precision multiplications. arXiv preprint arXiv:1412.7024 (2014)

13. Das, D., Mellempudi, N., Mudigere, D., *et al.*: Mixed precision training of convolutional neural networks using integer operations. arXiv preprint arXiv:1802.00930 (2018)

14. Elsen, E.: Optimizing rnn performance. `http://svail.github.io/rnn_perf/`, accessed: 2019-03-28

15. Graves, A., Liwicki, M., Fernandez, S., *et al.*: A novel connectionist system for unconstrained handwriting recognition. IEEE Trans. Pattern Anal. Mach. Intell. 31(5), 855–868 (2009), DOI: 10.1109/TPAMI.2008.137

16. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural Computation 9(8), 1735–1780 (1997), DOI: 10.1162/neco.1997.9.8.1735

17. Kalamkar, D., Banerjee, K., Srinivasan, S., *et al.*: Training google neural machine translation on an intel cpu cluster. In: CLUSTER (2019 (to appear))

18. Karpathy, A.: The unreasonable effectiveness of recurrent neural networks. `http://karpathy.github.io/2015/05/21/rnn-effectiveness/`, accessed: 2019-03-28

19. Namin, A.H., Leboeuf, K., Muscedere, R., Wu, H., Ahmadi, M.: Efficient hardware implementation of the hyperbolic tangent sigmoid function. In: ISCAS. pp. 2117–2120 (2009), DOI: 10.1109/ISCAS.2009.5118213

20. Powell, M.J.D.: Approximation theory and methods. Cambridge university press (1981)

21. Rivlin, T.J.: The Chebyshev polynomials (Pure and Applied Mathematics). Wiley-Interscience (1974)

22. Tommiska, M.T.: Efficient digital implementation of the sigmoid function for reprogrammable logic. IEE Proceedings - Computers and Digital Techniques 150(6), 403–411 (2003), DOI: 10.1049/ip-cdt:20030965

23. Wen, T., Gasic, M., Mrksic, N., *et al.*: Semantically conditioned lstm-based natural language generation for spoken dialogue systems. In: EMNLP. pp. 1711–1721 (2015), DOI: 10.18653/v1/D15-1199

24. Wu, Y., Schuster, M., Chen, Z., *et al.*: Google's neural machine translation system: Bridging the gap between human and machine translation. CoRR abs/1609.08144 (2016), `http://arxiv.org/abs/1609.08144`