Optimizing Deep Learning LSTM Topologies on Intel Xeon Architecture

Kunal Banerjee, Evangelos Georganas, Dhiraj D. Kalamkar, Alexander Heinecke
Intel Corporation
Email: {**kunal.banerjee, evangelos.georganas, dhiraj.d.kalamkar, alexander.heinecke**}**@intel.com**

ISC 2019: Research Poster
June 16 – 20, 2019

Long Short-Term Memory (LSTM)

- LSTM is a type of recurrent neural network (RNN) which is well-suited for processing temporal data
- Unlike traditional RNN, LSTM can handle exploding and vanishing gradient problems encountered during neural network training
- LSTM has found applications in language translation, text generation, handwriting recognition and image captioning among many others
- Operations in the forward pass of an LSTM cell

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

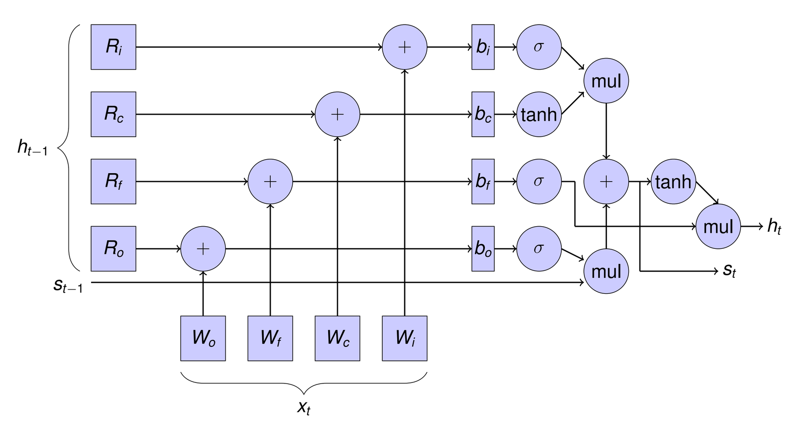

Figure 1: A diagram of an LSTM cell

Typical implementation of LSTM

- ☞ Perform two large GEMMs (**W** $*$ **x** and **R** $*$ **h**) or one *larger* GEMM (concatenated **WR** with concatenated **xh**)
- ✓ Easy to implement – leverage vendor-optimized GEMM
- × Weight reuse relies on how the GEMMs are parallelized and hence may be sub-optimal for GEMMs stemming from small minibatch size
- × Element-wise operations are exposed as bandwidth-bound kernel (vs in-cache reuse of the GEMM outputs)

Our implementation of LSTM

- ☞ Adopt a "dataflow" based approach for optimizations
- 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, transform the inputs in blocked format:
  - Input: $[T][N][C] \rightarrow [T][N/B_N][C/B_C][B_N][B_C]$
  - Hidden: $[T][N][K] \rightarrow [T][N/B_N][K/B_K][B_N][B_K]$
  - Weights: $[C][4K] \rightarrow [4K/B_K][C/B_C][B_C][B_K]$
  - Recurrent Weights: $[K][4K] \rightarrow [4K/B_K][K/B_K][B_K][B_K]$
  - B_N, B_C and B_K are blocking factors for N, C and K respectively
- Perform computation with fused time steps
  - Amortize cost of blocking
  - Optimized weight gradient computation
- Also, allow blocked inputs / weights to be passed directly from framework
  - Useful when performing one time step at a time
- Use JIT batch-reduce GEMM kernels
  - Implement optimized blocked GEMM
  - Implement fused kernel for elementwise operations (i_t, f_t, o_t, c_t, s_t, h_t)
    - Using Intel AVX512 intrinsics to vectorize
    - Use the Intel Short Vector Math Library (SVML) for fast tanh and sigmoid
  - Once a block of GEMM is computed, apply element-wise operations on it while hot in cache
- Our LSTM operators are thread-library agnostic (can use any of pthreads, OpenMP, C++ threads, Cilk, TBB, etc.)
- ☞ Same optimization principles applied to backward and weight update passes
- ☞ Our code is available through LIBXSMM at [9]

Integration into TensorFlow

- **XsmmFusedLSTM:** Implemented a wrapper in TensorFlow similar to LSTMBlockFusedCell
  - Single TensorFlow Op performing all time steps
  - Best for performance but may require significant source code change
- **XsmmLSTMCell:** A wrapper in TensorFlow compatible with BasicLSTMCell to perform single time step
  - Allows use of RNNCell wrappers like MultiRNNCell, DeviceWrapper, DropoutWrapper and ResidualWrapper
  - Allows easy replacement inside application code where fused cell is not used, e.g. GNMT
- **Weights:** Uses same layouts as in TensorFlow LSTMCell
  - Optimizes block transpose when using XsmmLSTMCell
  - Transpose happens outside time step loop when using dynamic_rnn

Experimental Setup

All the experiments and measurements are conducted over following hardware / software configuration

- **Machine:** Single socket Xeon Platinum 8180 with 28 Cores (3+ TFLOPS peak), NVIDIA K40m (4+ TFLOPS peak, [6])
- **MKL-DNN:** from github (commit 3439371) compiled with icc 19.0.0.117
- **LIBXSMM:** compiled with icc version 18.0.0 (we observed slowdown with latest icc/SVML version)
- **Stock Tensorflow w/o MKL:** v1.12.0 installed using "pip install tensorflow"
- **Tensorflow with MKL:** v1.12.0 compiled using gcc 8.3.0 with "-config=mkl"
- **GNMT:** NMT + GNMT attention (8 layers) with Minibatch: 168, inter_op_threads: 1, intra_op_threads: 28

GNMT end-to-end training with TensorFlow

- First, with few lines of source code change, we replaced BasicLSTMCell code by XsmmLSTMCell (XsmmLSTM)
- Then, we replaced unidirectional encoder layers with XsmmFusedLSTM layers (+Fused Encoders)
- Switching to the Fused Cell for decoders is subject to future due to Tensorflow's decoder implementation.
- For 8-layer German-to-English model, Perplexity and Gradient Norm of our implementation follows closely with reference run and we achieved similar BLEU score to reference version for 2-layer Vietnam-to-English translation
- Overall, we achieved 1.9× training speed up compared to original TensorFlow code for 8-layer German-to-English translation model exceeding Nvidia K40m performance
- Major benefits come from improved efficiency for forward pass GEMMs (1.5x speed up) and 12× reduction in cost for elementwise operations (from 30% to 2.5%). Out of 24% of backward/update elementwise operations, single BiasAddGrad takes about 16% of time which reduces to less than 1% after optimization

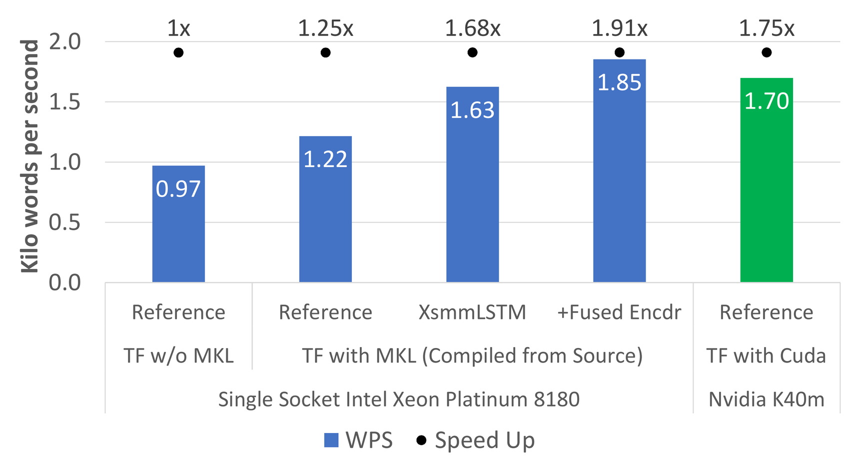

Figure 2: GNMT 8-layer Performance (with Turbo Enabled)

In a similar implementation "OpenSeq2Seq" NVIDIA V100 (5x faster than one Xeon 8180) achieves 4.8x more throughput

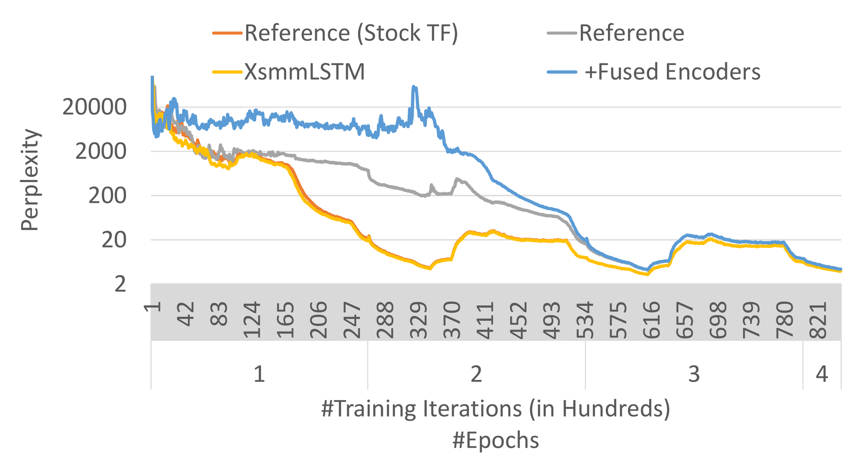

Figure 3: GNMT Convergence: Perplexity

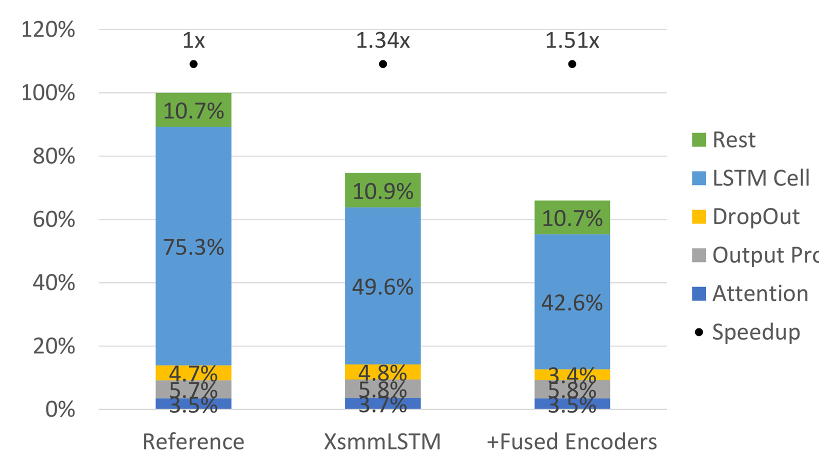

Figure 4: GNMT 8-layer: Overall Time Breakup

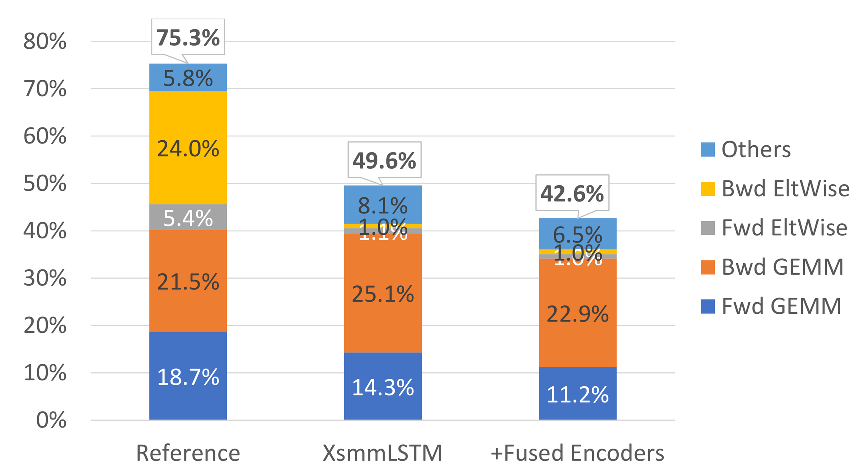

Figure 5: GNMT: Time Spent inside LSTM Cell

LSTM cell efficiency

- 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
- To demonstrate that our LSTM cell offers best-in-class performance, we not only compare to Tensorflow end-to-end but also to the MKL-DNN LSTM cell which is not available in Tensorflow at the time of this writing

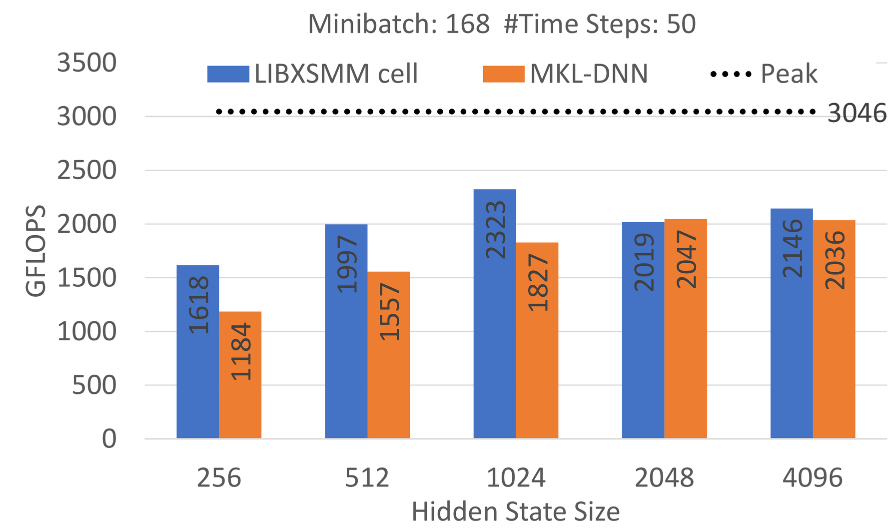

Figure 6: Forward pass results, Turbo disabled for stability

- LIBXSMM cell is up to 1.4× faster than MKL-DNN LSTM forward pass
- For large hidden state sizes, the two approaches exhibit similar performance
  - GEMM has cubic complexity while element-wise operations quadratic → for large sizes the element-wise operations/bandwidth overhead are less emphasized

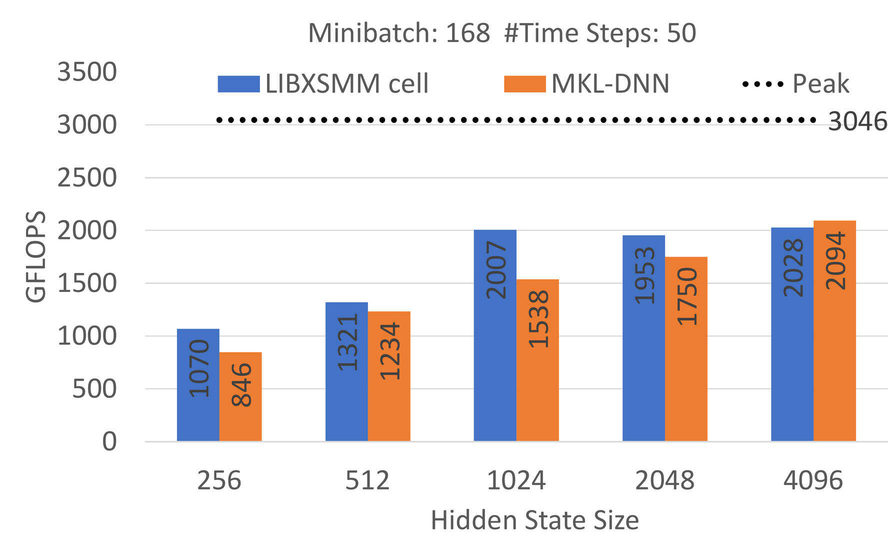

Figure 7: Backward/weight update pass results, Turbo disabled

- LIBXSMM cell is up to 1.3× faster than MKL-DNN LSTM backward/weight update pass

Summary

- Implementation of LSTM cell using a "dataflow" approach instead of large GEMMs
  - Maximize locality, weight reuse
  - Fuse element-wise operations
- For small/medium sized problems, our implementation of LSTM forward pass is up to 1.4× faster than the MKL-DNN cell, while for backward/weight update it is up to 1.3× faster
- For large weight matrices the two approaches have similar performance
  - Cubic GEMM scaling VS quadratic elementwise scaling
  - This conclusion may change with GEMM accelerated hardware
- Dataflow approach is well suited for CPUs
  - Coarse-grained parallelization and better locality control
- Modified TensorFlow which invokes our LSTM cell implementation is shown to perform end-to-end training attaining identical BLEU score and in as many iterations as original TensorFlow CPU implementation
- A speed up of 1.9× is achieved using our LIBXSMM LSTM cell over original TensorFlow implmentation for 8-layer German-to-English translation model training

Current Research

- Our LSTM cell also supports *bfloat16* – a new datatype introduced by Intel – however, further tuning is needed to expose its full potential
- Other than LSTM, we have also implemented vanilla RNN and Gated Recurrent Unit (GRU) (available online on github); we intend to experiment with these variants of RNN and report their performance benefits on neural network training/inference
- Evaluating how and if the proposed JIT batch-reduce GEMM kernel can be used on GPU or deep learning focused architectures

References:

[1] Sepp Hochreiter, Jurgen Schmidhuber. Long Short-Term Memory, Neural Computation 9(8): 1735–1780, 1997.
[2] Alexander Heinecke, Greg Henry, Maxwell Hutchinson, Hans Pabst. LIBXSMM: Accelerating small matrix multiplications by runtime code generation, SC 2016: 981–991.
[3] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le et al. Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation, CoRR abs/1609.08144, 2016.
[4] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen et al. TensorFlow: A System for Large-Scale Machine Learning, OSDI 2016: 265–283.
[5] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, Yoshua Bengio. Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling, CoRR abs/1412.3555, 2014.
[6] GNMT – TensorFlow Neural Machine Translation Tutorial, `https://github.com/tensorflow/nmt`
[7] MKL-DNN – Intel® Math Kernel Library for Deep Neural Networks, `https://github.com/intel/mkl-dnn`
[8] TensorFlow – An Open Source Machine Learning Framework for Everyone, `https://github.com/tensorflow/tensorflow`
[9] LIBXSMM – Library targeting Intel Architecture for specialized dense and sparse matrix operations, and deep learning primitives, `https://github.com/hfp/libxsmm`