EcoRNN: Efficient Computing of LSTM RNN on GPUs

Extended Abstract

Bojian Zheng
M.Sc. Candidate, Department of Computer Science
University of Toronto
bojian@cs.toronto.edu

Advisor: Gennady Pekhimenko
Assistant Professor, Department of Computer Science
University of Toronto
pekhimenko@cs.toronto.edu

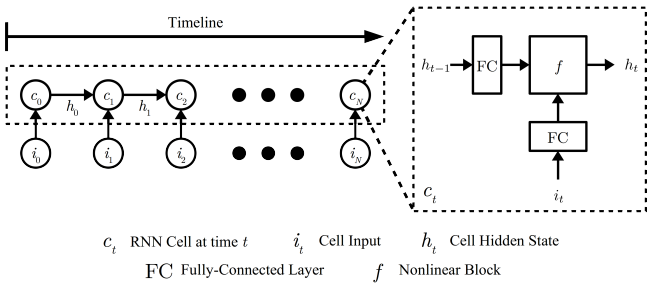

Figure 1: Left: A single-layer LSTM RNN that scans through an input sequence. Right: A zoom-in view of an LSTM cell. Both diagrams have been greatly simplified.

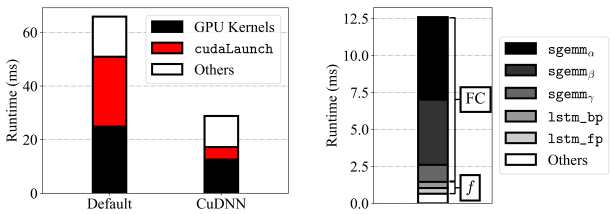

Figure 2: (a) Runtime profile comparison between *Default* and *cuDNN*. *Default* has 5× more cudaLaunch overhead compared with *cuDNN*. (b) Runtime breakdown by GPU kernels in *cuDNN*. All kernel names have been abbreviated.

1 INTRODUCTION

Long-Short-Term-Memory Recurrent Neural Network (LSTM RNN [7], Figure 1) is a state-of-the-art model for analyzing sequential data. Current implementations of LSTM RNN in machine learning (ML) frameworks usually either lack performance or flexibility (i.e. the ability to modify existing computation of LSTM RNN). For example, default implementations (which we will further refer to as *Default*) in Tensorflow [1] and MXNet [3] invoke many tiny GPU kernels, leading to excessive overhead in launching GPU threads (a.k.a. cudaLaunch, Figure 2a). Although *cuDNN* [5], NVIDIA's deep learning library, can accelerate performance by around 2×, it is closed-source and inflexible, hampering further research and performance improvements in frameworks, such as PyTorch [17], that use *cuDNN* as their backend [6]. In this work, we introduce a new RNN implementation called *EcoRNN* that is significantly faster than the open-source implementation in MXNet and is competitive with the closed-source cuDNN. We show that applying data layout optimization, with other techniques that are shown in Appleyard et al.'s work, can give us a maximum performance boost of 3× over MXNet default and 1.5× over cuDNN implementations. We integrate *EcoRNN* into MXNet Python library and open-source it to benefit ML practitioners. We also find that ML compilers [19, 4, 18] can be helpful in making the frontend programming interface more flexible by automatically configuring the data layout to be optimal, which is not possible in cases other than LSTM RNN since it has the property of "optimizing once and for all".

2 KEY OBSERVATIONS

We make the following observations during our optimization of LSTM RNN compututation on the GPUs.

Observation 1. The runtime bottleneck of LSTM RNN is fully-connected (FC) layers. We obtain the runtime breakdown of *cuDNN* GPU kernels using *nvprof* [15], and the result is shown in Figure 2b (due to the fact that *Default* slices the computation of "f" block in Figure 1 into small pieces, its result is difficult to interpret). Figure 2b shows that more than 85% of the time spent on compute has been allocated to matrix multiplies (sgemm is the name for single-precision matrix multiply kernels in cuBLAS library [12]). The annotations beside the stacked bar in Figure 2b group GPU kernels together according to their counterpart in Figure 1, which explains why FC layers in LSTM RNN should be the top candidate for optimization.

Observation 2. Data layout optimization can speed up FC layers. Data layout optimization is a technique that originates from compiler research [9]. The idea behind data layout optimization is that changing data layout (usually from row-major to column-major or vice versa) can result in better locality in the data access pattern, which will further lead to higher cache hit rate, faster memory accesses, and, eventually, better runtime performance. We discover that two computations, $Y = XW^T$ and $Y^T = WX^T$ (Figure 3a), that are mathematically the same can have different cache hit rates. Figure 3b shows an example of such comparison where $X : [64 \times 512]$ and $W : [2048 \times 512]$ (this mimics the FC layers of an LSTM RNN that has batch size and hidden dimension equal to 64 and 512 respectively). We observe that $Y^T = WX^T$ is almost twice as fast as $Y = XW^T$ under this parameter setting, and the reason is that the former has better cache hit rate (50% vs. 10% of latter). Therefore, it can feed data faster into compute units, and thus spend more time in compute rather than waiting for data to arrive from main memory.

3 PRELIMINARY RESULTS

All the experiments are done on a single machine with Intel®Core™i5-3570 [8] CPU and Titan Xp [16] GPU. We have been using CUDA 8.0 [14] toolkit and cuDNN 6.0 [13] for our experiments. To provide fair comparison against *Default* and *cuDNN*, we integrate our implementation of *EcoRNN* into MXNet ver. 0.12.1. We test all three

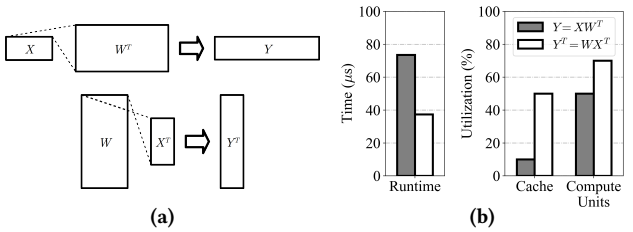

Figure 3: (a) $Y = XW^T$ (top) vs. $Y^T = WX^T$ (bottom). Both are doing the same amount of computation. (b) Runtime (left) and hardware utilization (right) comparison between $Y = XW^T$ and $Y^T = WX^T$.

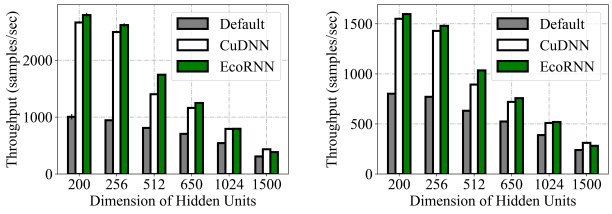

Figure 4: Training throughput comparison on PTB (left) and Wikitext-2 (right) dataset.

backends end-to-end (i.e. all overheads have been included) with the MXNet language modeling benchmark on the PennTreeBank (PTB) [10] and Wikitext-2 [11] dataset. Figure 4 shows the results of training throughput comparison. We observe that *EcoRNN* is always significantly better than *Default* and in most cases better than *cuDNN*. Even in a few cases where *cuDNN* slightly outperforms *EcoRNN*, the performance difference is below 20%.

4 OPTIMIZING ONCE AND FOR ALL

The trade-off between performance and flexibility is ubiquitous in high-performance-computing. With the advent of machine learning compilers [19, 4, 18], it is made possible to perform dynamic code optimization under a easily programmable frontend interface, therefore achieving both high performance and flexibility simultaneously. We notice that the optimization of LSTM RNN exhibits the property of "once and for all" – optimizations that are performed for one single cell can be generalized immediately to all other cells of different layers and time steps. Therefore, data layout optimization, which is a NP-complete problem in generic settings, can be reduced to a binary problem in the case of LSTM RNN. We aim to investigate the use of ML compilers in RNN training, which can help us dynamically select between the transposed and legacy layout depending on the hyperparameter settings at runtime. Figure 5 demonstrates how it works – at the frontend, machine learning programmers specify the hyperparameters that they are hoping to use, our microbenchmark then quickly does runtime comparison between different implementation backends (i.e. *Default*, *cuDNN*, and *EcoRNN*) in milliseconds, the best of which will be used in actual training that lasts for hours.

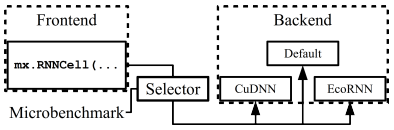

Figure 5: Runtime optimization tool that selects between different implementation backends depending on the decision made by the microbenchmark.

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, et al. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/

[2] Jeremy Appleyard, Tomás Kociský, and Phil Blunsom. 2016. Optimizing Performance of Recurrent Neural Networks on GPUs. *CoRR* abs/1604.01946 (2016). arXiv:1604.01946 http://arxiv.org/abs/1604.01946

[3] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *CoRR* abs/1512.01274 (2015). arXiv:1512.01274 http://arxiv.org/abs/1512.01274

[4] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Q. Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: End-to-End Optimization Stack for Deep Learning. *CoRR* abs/1802.04799 (2018). arXiv:1802.04799 http://arxiv.org/abs/1802.04799

[5] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cuDNN: Efficient Primitives for Deep Learning. *CoRR* abs/1410.0759 (2014). arXiv:1410.0759 http://arxiv.org/abs/1410.0759

[6] Hardik Goel. 2017. Add Peephole connections for LSTMs? https://github.com/pytorch/pytorch/issues/630

[7] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Computation* 9, 8 (Nov. 1997), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

[8] Intel. 2012. Intel® Core™ i5-3570 Processor (6M Cache, up to 3.80 GHz). https://ark.intel.com/products/65702/Intel-Core-i5-3570-Processor-6M-Cache-up-to-3_80-GH

[9] Ken Kennedy and Ulrich Kremer. 1998. Automatic data layout for distributed-memory machines. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 20, 4 (1998), 869–916.

[10] Mitchell P. Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. 1993. Building a Large Annotated Corpus of English: The Penn Treebank. *Comput. Linguist.* 19, 2 (June 1993), 313–330. http://dl.acm.org/citation.cfm?id=972470.972475

[11] Stephen Merity. 2016. The wikitext long term dependency language modeling dataset.

[12] NVIDIA. 2017. cuBLAS Library v8.0.

[13] NVIDIA. 2017. cuDNN v6.0.

[14] NVIDIA. 2017. NVIDIA CUDA Toolkit v8.0.

[15] NVIDIA. 2017. Profiler User's Guide v8.0.

[16] NVIDIA. 2017. Titan Xp User Guide. http://www.nvidia.com/content/geforce-gtx/NVIDIA_TITAN_Xp_User_Guide.pdf

[17] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. (2017).

[18] Tensorflow. 2018. XLA Overview. (2018). https://www.tensorflow.org/performance/xla/

[19] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. *CoRR* abs/1802.04730 (2018). arXiv:1802.04730 http://arxiv.org/abs/1802.04730