Analysis of Tensorflow and PyTorch Hardware Accelerated LSTM Kernels

Kaushik Mahorker
UCLA - 805352314
kaushikm@ucla.edu

Amir Ali Yazdi-Nejad
UCLA - 204846462
amiraliyn@ucla.edu

Dennis Zang
UCLA - 704766877
dzang8055@ucla.edu

ABSTRACT

The increasing popularity of deep learning has brought the need for user friendly implementations of GPU accelerated training and inference. State of the Art libraries such as Tensorflow and PyTorch have been able to fill the usability gap of pure CUDA code. However, with the prioritization of usability, we tradeoff efficiency. In this paper, we measure the extent of these tradeoffs and benchmark multiple implementations of the same neural network. We focus on evaluating LSTM Kernels that are GPU accelerated across a myriad of parameters. We find that PyTorch is significantly faster than Tensorflow and about 53% slower than our DeepBench CUDA baseline. We also explore automatic mixed precision techniques and find them to be slightly faster for larger timesteps. Here is the link to our spreadsheet which contains our benchmarks: https://docs.google.com/spreadsheets/d/19gRckZok10r7niuxE-X11rL_XaIfOo0t-830xRnx8Zo/edit#gid=0

ACM Reference Format:
Kaushik Mahorker, Amir Ali Yazdi-Nejad, and Dennis Zang. 2018. Analysis of Tensorflow and PyTorch Hardware Accelerated LSTM Kernels. In *Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

State-of-the-art (SoTA) libraries have been providing convenience and allowing industry professionals to build production models quickly. However, by making the process more convenient, we are taking away efficiency. Since efficiency is having increasing importance as models get larger, we decided to setup multiple experiments to explore the relationship between Baidu DeepBench (a pure CUDA/C++ LSTM kernel implementation which we used as our baseline) with configured implementations of PyTorch and TensorFlow that mimic the execution pattern of DeepBench.

2 RELATED WORK

This project leverages the work of various deep learning libraries and their associated tools. Our exploration consists of Tensorflow [2], PyTorch [7] and Baidu DeepBench [3] implementations of an LSTM kernel. Tensorflow is an OSS library developed by Google with dataflow and differentiable programming across many tasks. It is primarily used for machine learning and neural network training. It is written in C++, but interfaced to users in Python. PyTorch is another OSS machine learning library similar to Tensorflow, developed by Facebook AI and written mostly in Python. PyTorch is found more in the research setting, whereas Tensorflow is found more in the production setting. Baidu DeepBench is an OSS library to benchmark operations that are important to deep learning. It's primary purpose is to examine which hardware provides the best performance on the basic operations used for neural networks. DeepBench is written in CUDA and leverages the cuDNN library as an accelerator. Under the hood, all these libraries use CUDA implementations to accelerate computations with the GPU. However, not all CUDA implementations are made the same and this work aims to explore this.

Other works [5] [4] also explore the efficiency of PyTorch and Tensorflow and their GPU utilization. [5] focuses on improving the GPGPU-Simulator to be able to track cuDNN information on memory usage, power and efficiency. They focus on evaluating the MNIST example and convolution kernels. While this approach does provide more information, these extra steps and potential pitfalls are avoidable by using built-in and tested profilers. Simulators have to built in a way that while capturing information, they do not skew it with their own overhead. [4] also evaluates the MNIST dataset and convolution kernels for hardware utilization, hardware temperature and execution time. Our work differs because of its primary focus on LSTM kernels rather than convolution. We also aggregate over many parameters and different combinations to provide a case wise analysis and recommendation of the implementation to use.

3 METHODS

We setup multiple experiments to explore the relationship between pure CUDA LSTM Kernel implementations and that of state of the art libraries. We consider Baidu Deepbench, a pure CUDA/C++ LSTM kernel implementation, as our baseline and configure PyTorch and Tensorflow implementations that mimic the execution pattern of Deepbench. We also leverage NVIDIA-Apex [6] to enable automatic mixed precision on Pytorch to see if we can get any performance benefits. All experiments were run in inference mode.

3.1 Baidu DeepBench

Baidu Deepbench is a reliable benchmarking software for many deep learning operations. Although it focuses on discerning which hardware is best suited for the task, we leverage it to discern which implementation is best for the hardware. Deepbench provides a C++ wrapper for easy definition of LSTM parameters and calls CuDNN under the hood. These operations are run synchronously and in order rather than batched, so we get the time elapsed for

Woodstock '18, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

the execution of a single LSTM cell. Deepbench doesn't provide support for benchmarking the execution of multiple LSTM layers in sequence, therefore our analysis is limited to a single LSTM layer. We extract the forward time for each combination of parameters as our baseline.

3.2 Tensorflow

Tensorflow is a state of the art open source neural networks library developed by Google Brain with the goal of democratizing deep learning for everyone. Tensorflow uses Keras under the hood to define and construct its neural network layers. We use the Keras LSTM models for our implementation. These models are batch major rather than time major like Deepbench and PyTorch, meaning that the batch size is the first dimension instead of the LSTM time steps. We choose to leave this setting as is because a typical user will likely prefer a batch major approach. In addition, Table 1 shows the various GPU kernels Tensorflow can use to accelerate LSTM execution. We perform experiments on all three modes.

Kernel Name	Description
TF Mode 0	Same as cuDNN library Kernel
TF Mode 1	More batches with many smaller dot products and additions
TF Mode 2	Small, but large batches with larger operations
TF AMP	Mixed precision casting ops followed by cuDNN library Kernel
PyTorch	Same as cuDNN library Kernel
PyTorch AMP	Mixed precision casting ops followed by cuDNN library Kernel

Table 1: GPU Accelerated LSTM Kernels used by SoTA Libraries

3.3 PyTorch

PyTorch is another state of the art open source deep learning library and is used more in the research setting, whereas Tensorflow is used in a production setting. PyTorch defines its own LSTM model and has a different implementation than Keras. PyTorch LSTMs are time major, but take inputs explicitly specified as three separate vectors: the input vector, hidden state vector and cell state vector. We configured these vectors to have the same dimensions as Tensorflow and Deepbench. As long as the Tensors are initialized on the GPU and cuda is specified, PyTorch will automatically use the cuDNN library implementation. In addition, PyTorch makes internal optimizations to launch the cuda kernel quickly [8] and have as little overhead as possible.

3.4 Automatic Mixed Precision

Apart from the standard PyTorch and Tensorflow implementations and built-in variations of LSTM kernels, we also explore the effects of using mixed precision tensors and models on execution time. Mixed precision training refers to the ability to train with half precision, i.e. 16-bit rather than 32-bit floats, and maintain the network accuracy achieved with single precision. Although we are focusing our evaluation on inference, we can still leverage mixed precision as the same forward pass is taken during inference as it is during training. We leverage NVIDIA-Apex [6] to enable Automatic Mixed Precision (AMP) in our PyTorch implementation. We were unable to find a similar approach for DeepBench. A few casting operations are executed on the tensors on the GPU before launching the cudnn_rnn kernel, but otherwise is the same sequence of operations as the original PyTorch implementation. For Tensorflow, we make use of the Keras dtype policy provided in the Keras mixed precision API [1] to enable AMP between tf.float16 and tf.float32.

4 EVALUATION

We leverage the implementations described in Section 3 to analyze the execution time and accelerator design decisions for kernels shown in Table 1. We control different LSTM parameters to observe the scalability of each approach. Specifically we tune the number of hidden elements, batch size and time steps. As Deepbench uses only a single LSTM cell in its implemenation, we keep that constant across all implementations. Our primary tracked metrics are overall execution time and kernel operation-wise execution time. We also leverage tensorboard to track GPU activity for Tensorflow runs. All experiments are run on the tetracosa UCLA server with a 12 GB TITAN V GPU.

Kernel Names	MPE
TF Mode 0 vs TF Mode 1	-1.822121106
TF Mode 0 vs TF Mode 2	-1.244951865
TF Mode 1 vs TF Mode 2	-0.1308454709
TF Mode 0 vs TF Mode 0 AMP	-0.1455642638
DeepBench vs TF Mode 0	-17.75424073
DeepBench vs PyTorch	-0.53045191
PyTorch vs PyTorch AMP	-0.6300867506
DeepBench vs PyTorch AMP	-1.304903692

Table 2: Comparing the execution time of different kernels (negative numbers mean the second kernel is slower than the first one)

4.1 Tensorflow Modes

When testing out our implementation of LSTM using Tensorflow, we need to note that Tensorflow has three modes of operations: mode 1, which specializes in a large number of small dot products and additions; and mode 2, which specializes in batching these operations into a smaller number of large operations (as shown in Table 1). From this, we would expect that mode 1 would do better for smaller inputs and smaller numbers of hidden layers, while mode 2 would do better for the opposite, as Tensorflow would make better use of the GPU's resources for parallel computation.

As we can see in Figures 1 and 3, this is definitely the case. Figure 1, which maps batch size to the forwarding time, shows mode 1 performing better than mode 2 for batch sizes smaller than equal to 16. However, as we increase the batch size even further, mode 1 falls behind mode 2 as the input size increases beyond a certain point, and falls behind significantly for a batch size of 256. This is somewhat expected, because as batch size increases, mode 2 would

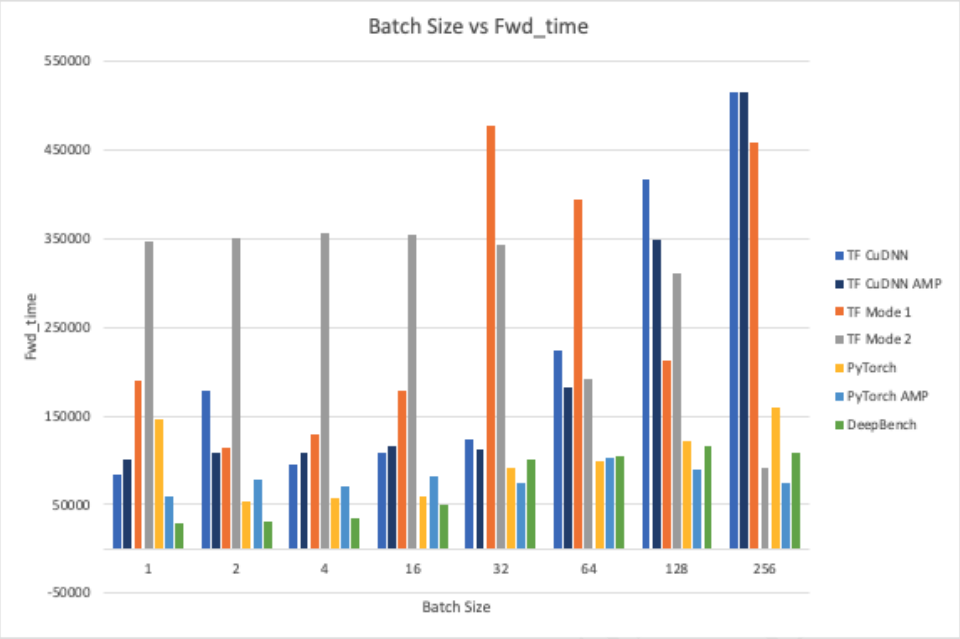

Figure 1: Comparing the execution time of different kernels based on their batch sizes

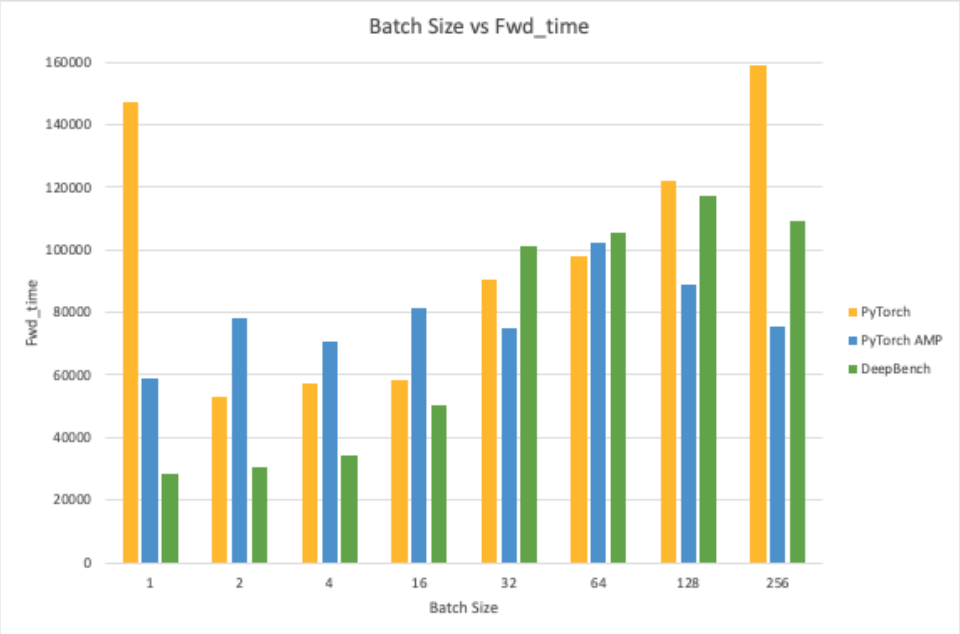

Figure 2: Comparing the execution time of PyTorch, PyTorch AMP and DeepBench kernels based on their batch sizes

have an advantage over mode 1 as it aggregates smaller operations into larger ones.

However, it is the opposite for figure 3, which shows mode 1 performing worse mode 2 for smaller time steps, before overtaking by mode 2 for time steps greater than equal to 1000. This is also expected, since if the input were to increase in size, then mode 1 would have a definite advantage over mode 2 in repeatedly feeding in input into the recursive neural network.

An interesting observation we made when we were testing Tensorflow under different modes of operation is that mode 1 spent about 72.4% of the sampled time within the kernel launch state (Figure 6), with mode 2 performing about the same. Because of this,

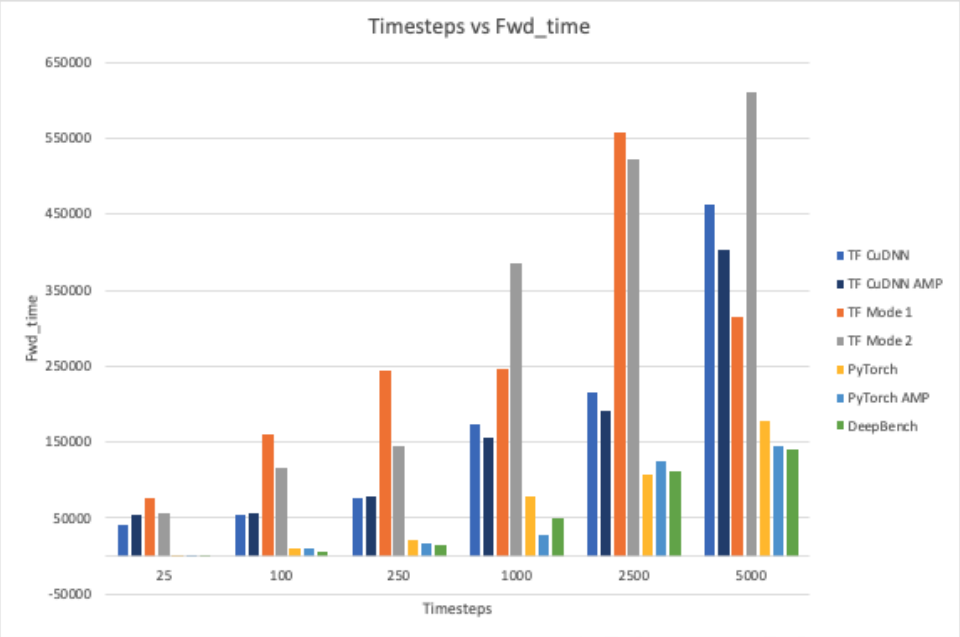

Figure 3: Comparing the execution time of different kernels based on their timesteps

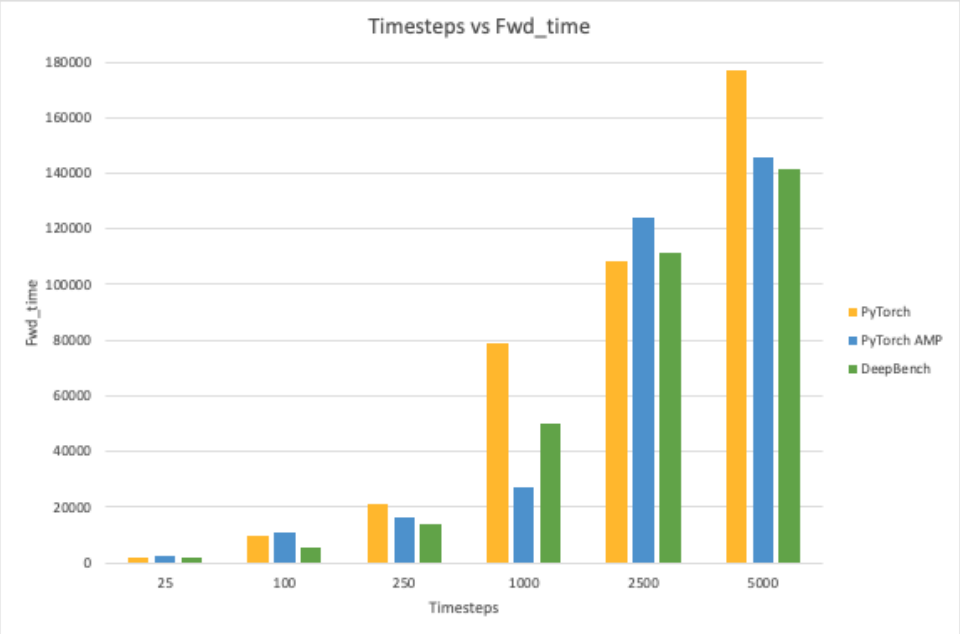

Figure 4: Comparing the execution time of PyTorch, PyTorch AMP and DeepBench kernels based on their timesteps

we modes 1 and 2 consistently performed worse than cuDNN, so in our case, there is almost no reason to prefer modes 1 and 2 over any other LSTM implementation.

4.2 PyTorch

As described earlier, our PyTorch implementation leverages the cuDNN kernel to accelerate the forward pass computation. Table 2 shows that PyTorch achieves execution times that are about 50% slower than DeepBench. Although the same GPU accelerator is used, factors such as kernel launching and simply the overhead of

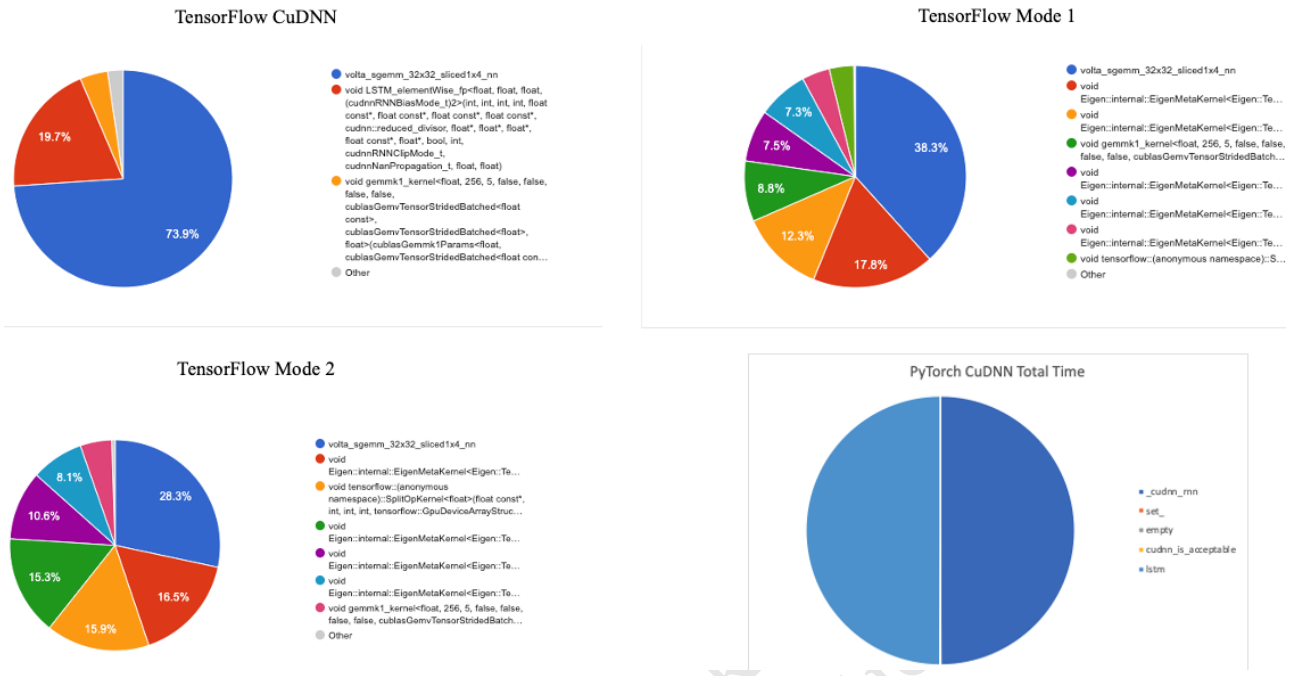

Figure 5: Comparing the execution time of the top few functions of TensorFlow Mode 0, TensorFlow Mode 1, TensorFlow Mode 2 and PyTorch

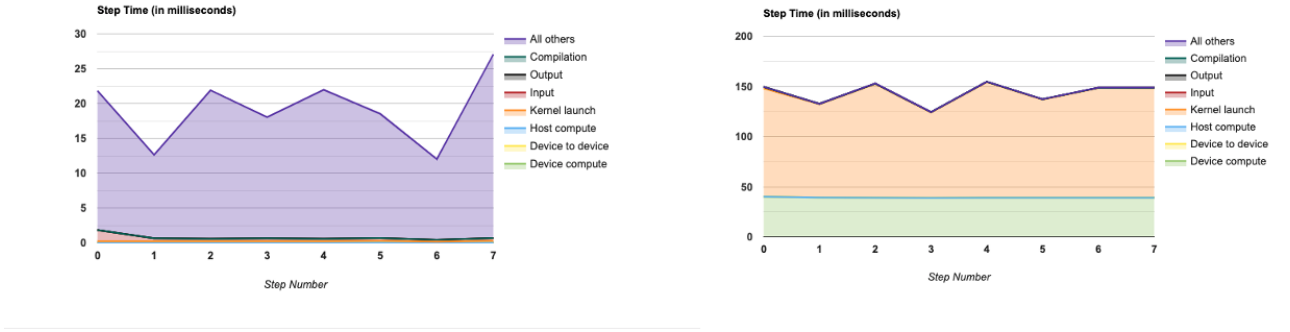

Figure 6: Comparing the execution time of the kernel launch in TensorFlow Mode 0, TensorFlow Mode 1

python as a programming language causes execution slowdowns. Figure 2 shows that for larger batch sizes, PyTorch is actually faster on average than DeepBench signifying that PyTorch is more resistant and scalable to increasing batch size than DeepBench. This relationship breaks when grouping execution time by the number of timesteps. Figure 4 reveals PyTorch's overhead for increasing timesteps, yet it is does not reach the 2x slowdown we observed on average.

From these results we can infer that timesteps are the bottleneck in determining which implementation to pick. PyTorch provides easy access to enabling cuda kernels and using cudnn implementations of the LSTM kernel. Although there is a slight overhead, for tasks with a smaller number of timesteps and larger batches, PyTorch is definitely worth the tradeoff. Eventhough larger timesteps and fewer batches merit the use of the more efficient pure CUDA implementation, deep learning tasks are generally large and the batch size is a tunable parameter at training time. PyTorch ensures a scalable implementation, where if down the line a larger batch size is need for less noisy gradients, it can prove to be faster than the pure CUDA implementation.

4.3 PyTorch vs. Tensorflow

Assessing the kernels within each of these libraries, we can make conclusions about the implementation to use on a case by case basis. However, when we have the freedom to pick between multiple implementations and libraries starting from scratch, it is also important to understand the tradeoffs between the SoTA libraries. For now we will only consider the standard use case on non-AMP optimized code. From an MPE perspective, Table 2 shows that Tensorflow Mode 0 is significantly slower than PyTorch. At first this result was surprising and we had questioned our implementation and whether it was actually using the GPU as we expected to see similar overheads between PyTorch and Tensorflow. Both are python based interfaces and leverage the cuDNN LSTM kernel. Double checking our results and doing further research, we found that others running benchmarks see similar results.

Figures 1 and 3 compares all the kernels listed in Table 1 and shows that Tensorflow consistently has the slowest execution times across all implementations. However, there are practical benefits that Tensorflow offers that still make this slowdown bearable for some use cases. Tensorflow is equipped with tensorboard and is a SoTA tool used to debug exploding and vanishing gradients and other training failures. It also profiles the GPU and kernel by operation, providing feedback for where optimizations can be made.

Figure 5 offers a potential explanation for the drastic slowdown of Tensorflow Mode 0. This shows a pie chart of all the GPU kernel operation execution times that each implementation took in its lifetime. Tensorflow has many more operations that are invoked than that of PyTorch. PyTorch only invokes the lstm kernel and cudnn_rnn operation and thus is only bound by these operations. On subsequent runs, Tensorflow will have many more operations to perform and different input sizes and parameters may affect the individual operation times causing more variance and dependencies in its execution pattern.

4.4 Automatic Mixed Precision

We successfully implemented Automatic Mixed Precision (AMP) for both PyTorch and Tensorflow. Profiling these AMP implementations shows extra kernel operations for casting input tensors into 16-bit float tensors. We evaluate and analyze these AMP approaches on the same set of parameters as the non AMP approaches.

4.4.1 PyTorch. Although at initial glance, the PyTorch AMP approach shows a 63% slowdown on average compared to the non-AMP PyTorch implementation, and is 1.3x slower than the Deepbench baseline, these measures are not consistent across all parameters. Taking a closer look, we observe batch-wise and timestep-wise aggregations of the execution times in Figures 2 and 4. There is no apparent relationship of execution time and batch size as at times it is faster then DeepBench and at others it is slower. For timesteps, the AMP is faster than PyTorch consistently and is significantly faster for larger timesteps. Looking at the raw data, for the set of parameters where PyTorch is faster than DeepBench, PyTorch AMP maximizes its value. Larger hidden sizes show improvements of >40% when compared to DeepBench. From these observations we can deduce that for larger parameters AMP is especially valuable, but for smaller parameter combinations, the bottleneck switches to type casting operations that cause mixed results.

4.4.2 Tensorflow. The Tensorflow AMP approach, at first, doesn't show too much of a significant difference compared to cuDNN if we are to compare the average of all forwarding times with respect to each batch size and each number of time steps, regardless of number of hidden layers and. However, if we are to observe the forwarding times directly for larger numbers of hidden layers (about 512 to 1024) and time steps, we can see that Tensorflow AMP does end up consistently about 10-15% faster than cuDNN for the largest input sizes. In addition, as we can see in Figure 3, as the number of time steps increase, Tensorflow AMP gradually overtakes cuDNN in performance.

An important observation we would like share is that, for smaller input sizes, the forwarding times for AMP is greater than that of simply running cuDNN. This is to be expected, as AMP does impose the extra overhead of performing tf.float16-tf.float32 conversions between the host CPU and the GPU. From the above, we can see that, if the input size is too large compared to the number of hidden layers, then the overhead from the conversions will overshadow the actual execution time.

Another factor to consider would be the number of time steps. As we can see in the data, for a given number of hidden layers and disregarding batch size, each specific number of time steps corresponds to a specific range of MPEs. From what we can see, increasing the number of time steps actually increases the performance of AMP, since execution time is directly proportional to it.

From the above, we observe that the overhead from the tf.float16-tf.float32 conversions is proportional to (time steps), while the computation time in the GPU is proportional to (hidden layers) x (time steps) (assuming the GPU has sufficient resources to work with the batches in parallel). So, in order to minimize the overhead from float conversions to make the most use of the tensor cores, it is advised to do so for larger number of hidden layers and time

steps so the overall percentage of the overhead compared to the in-GPU execution time would be minimized.

5 FUTURE WORK

Our analysis of cuDNN, Tensorflow, and Pytorch can be further extended to implementations of other hardware platforms like TPUs and other GPUs that use tensor cores. The TPU, an ASIC recently developedby google for deep learning, works with Google's version of Tensorflow, and can boost performance by making use of matrix processors (like tensor cores) in a systolic array architecture.

Although we have tried to use Tensorflow for AMP, we can take extra steps to verify its validity by testing on other GPUs and other machines. We could also try using different implementations of Tensorflow (like with Nvidia's NGC).

While we focused more on performance in our experiment, another performance benchmark we can also explore is energy consumption. In a future experiment, we could attempt to benchmark memory bandwidth/movement and estimate how much energy is used. Energy is also an increasingly important metric to track as more of these deep learning operations are being run in large data centers and coupled with other energy consuming tasks.

6 CONCLUSION

SoTA libraries offer great benefits of convenience and usability, but as our analysis shows do come with the tradeoff of efficiency. PyTorch and Tensorflow will continue to be the primary means of deep learning research and production, but understanding their intricacies can help us make the right design decisions. Our analysis explore different GPU accelerated LSTM kernels implemented by these libraries and plots them against a baseline CUDA implementation. We find that PyTorch is significantly slower than Tensorflow for most cases and that applying mixed precision does help for larger hidden sizes and timesteps. While work is being done to bring these SoTA libraries closer to optimal time, it is unlikely that they will keep up with cuDNN optimizations. Therefore, it is important for each developer to understand the GPU kernels used and tradeoffs in execution times when using these high-level libraries.

7 STATEMENT OF WORK

Kaushik worked on setting up Tensorflow, PyTorch and Tensorboard on tetracosa as well as implementing and evaluating the initial Tensorflow and PyTorch LSTMs. He also worked on the PyTorch AMP implementation.

Amir worked on gathering the baseline DeepBench metrics and putting together the graphs and tables. He also helped with the initial LSTM implementations.

Dennis worked on implementing and evaluating Automatic mixed precision for Tensorflow and PyTorch. He also worked on researching future extensions to the work.

REFERENCES

[1] [n.d.]. Mixed precision : TensorFlow Core. https://www.tensorflow.org/guide/keras/mixed_precision

[2] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 265–283. https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf

[3] Baidu-Research. 2019. baidu-research/DeepBench. https://github.com/baidu-research/DeepBench

[4] Felipe Florencio, Thiago Silva, Edward Ordonez, and Methanias Júnior. 2019. Performance Analysis of Deep Learning Libraries: TensorFlow and PyTorch. *Journal of Computer Science* 15 (05 2019). https://doi.org/10.3844/jcssp.2019.785.799

[5] Jonathan Lew, Deval Shah, Suchita Pati, Shaylin Cattell, Mengchi Zhang, Amruth Sandhupatla, Christopher Ng, Negar Goli, Matthew D. Sinclair, Timothy G. Rogers, and Tor Aamodt. 2018. Analyzing Machine Learning Workloads Using a Detailed GPU Simulator. arXiv:1811.08933 [cs.DC]

[6] Nvidia. 2020. NVIDIA/apex. https://github.com/NVIDIA/apex

[7] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.). Curran Associates, Inc., 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[8] PyTorch Team. [n.d.]. PyTorch. https://pytorch.org/blog/optimizing-cuda-rnn-with-torchscript/