Reducing Synchronization Overhead for Persistent RNNs

Senior Honors Thesis

Qinjun Jiang
Advisor: Matthew D. Sinclair
Department of Computer Sciences
University of Wisconsin-Madison
qjiang47@wisc.edu

Abstract

Recurrent Neural Networks (RNNs) are an important category of Deep Neural Network (DNN) that are widely used in speech recognition, sentiment analysis, and many other important applications. RNNs often use GPUs to harness the available parallelism to improve performance. However, their efficiency is hindered by sequential data dependencies in each timestep on outputs from previous timesteps, which makes it difficult to efficiently utilize GPUs and other accelerators for RNNs. In an effort to overcome this inefficiency, previous work introduced Persistent Recurrent Neural Networks (PRNNs), which improve efficiency by exploiting kernel fusion and the GPU's memory hierarchy to reuse network weights over multiple timesteps. Thus, PRNNs have been shown to outperform regular RNNs. However, since PRNN condense the entire RNN, which normally spans hundreds of GPU kernels, into a single large kernel with multiple phases, they require explicit synchronization at the end of each phase within the kernel. Unfortunately, synchronization in GPUs, such as global barriers, is extremely inefficient. This problem worsens for more complex RNNs such as Gated Recurrent Unit (GRU) and Long Short-Term Memory (LSTM) which require additional global synchronization to synchronize their additional sub-computations. Moreover, prior open-source work only added persistency Vanilla RNNs, whose usage is very limited in the real world. To extend the benefits of persistency to more applications, we implement Persistent Gated Recurrent Units (PGRU) to enable further research opportunities on PRNNs, including reducing synchronization overhead for both PRNN and PGRU.

1 Introduction

Recurrent Neural Networks (RNNs) are widely used in real world applications. For example, Google Translate uses RNNs for machine translation. Other representative RNN applications include speech recognition [5], sentiment analysis [10] [11], and even music generation [12] [13]. RNNs have significant data parallelism because all neural cells in a hidden layer perform the same computation on different data. Therefore, RNNs are well suited to run on hardware architectures that can process data in parallel. Thus, GPUs are well suited to run RNNs because, compared to the small number of cores in CPU, GPUs have hundreds of cores that RNNs can exploit [14]. However, RNN's sequential nature also cause data dependencies across RNN timesteps (a timestep is a single occurrence of an RNN cell that produces one hidden state, as in Figure 3): each RNN timestep must wait for the output of previous timestep in order to proceed. To respect this dependency, RNN implementations on GPUs utilize either implicit or explicit synchronization between timesteps, which hurts the efficiency of RNNs on GPUs.

To respect the data dependencies between timesteps, the most common RNN implementations on GPU utilize many small kernels (discussed further in Section 2.2). These small kernels each incur overhead from kernel launches. Moreover, each kernel must load data from the CPU to the GPU's global memory, and then move this data into the GPU's fast, local registers and shared memory (which are used to store the data in order to improve efficiency). Furthermore, once the kernel is complete, the data must be written out of local registers and shared memory, and written back to the GPU global memory. Thus, GPU RNN implementations have significant overheads and, due to their popularity and wide use, it is imperative to design more efficient RNN implementations.

Persistent Recurrent Neural Networks (PRNNs) seek to improve the RNN performance on GPU by utilizing kernel fusion and increasing data reuse. To avoid the overheads from repeated kernel launches and data transfers, PRNN *fuses* the kernels into a single, large kernel. This approach, also known as kernel fusion, enables data to be persistently stored in the fast, local memory – thus enabling reuse over multiple timesteps. As a result, prior work has shown that PRNN substantially improves computational throughput by designing persistent computational kernels that exploit GPU's memory hierarchy to reuse network weights over multiple timesteps [1]. CUDA also has its own PRNN implementation [15], which includes support for more complex RNNs, but it is not open sourced. Thus, to further investigate and experiment on PRNNs, we choose to extend Baidu's open source implementation [4].

PRNN's use of kernel fusion also removes the implicit synchronization points at kernel boundaries. Thus, to respect the data dependencies between timesteps, PRNN adds explicit global synchronization between timesteps. Unfortunately, synchronization in GPUs, such as global barriers, is extremely inefficient. Although prior work has explored ways to redesign and improve GPU synchronization [2][3][6], significant inefficiencies still remain. In GPUs, global

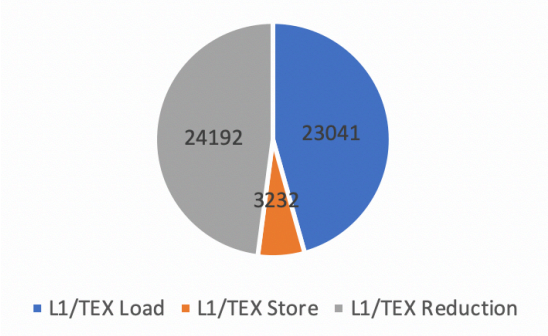

Figure 1a: Breakdown of memory accesses for Vanilla RNN training.

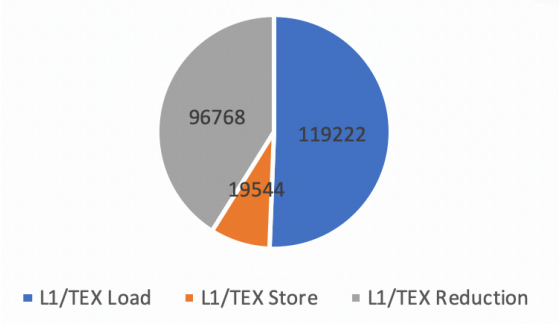

Figure 1b: Breakdown of memory accesses for Vanilla RNN inference.

synchronization is performed at the first level of the memory hierarchy common to all the synchronizing threads (usually the L2 cache). Since GPU L2 cache accesses take over 100 cycles and during global synchronization thousands of concurrent GPU threads are contending for a small number of synchronization variables, synchronization is often a serial bottleneck for PRNNs. For example, as shown in Figure 1, synchronization accesses make up 41% (inference) to 48% (training) of all memory accesses in GPU Vanilla PRNNs.

Furthermore, this problem worsens when using more complex RNNs such as Gated Recurrent Unit (GRU) [8][9] and Long Short-Term Memory (LSTM) [7]. GRU and LSTM use sophisticated filtering to retain information about previous timesteps longer and improve accuracy over Vanilla RNNs. However, to improve accuracy GRU and LSTM perform more sub-computations per timestep than Vanilla RNNs. As a result, their PRNN implementations require additional global synchronization (discussed further in Section 2.3). Since GRU and LSTM have similar overheads, but GRU is simpler, we focus on GRU in this work. However, since open source PRNN implementations only provide support for Vanilla RNNs, this requires extending them to provide support for a persistent GRU (PGRU) implementation. Thus, in this thesis, we extend an existing open source Vanilla PRNN to provide support for PGRU, which enables additional future research on optimizing synchronization for PRNNs.

The remainder of the report is organized as follows. In Section 2, we present background information on RNN, PRNN, and GRU. In Section 3, we discuss some findings when investigating the PRNN implementation. In Section 4, we discuss the implementation of the persistent version of GRU. In Section 5, we discuss future directions of this project. Finally, in Section 6, we conclude.

2 Background

2.1 RNN

RNNs are a major category of Deep Neural Network (DNN) that focus on sequence prediction tasks. Although there are many RNN flavors, we focus on Vanilla (the simplest) and GRU (which improves accuracy over Vanilla). At a high-level, as shown in Figure 2, RNNs exhibit a recurrent structure that processes sequential inputs one at a time. At each timestep, an RNN model takes not only a current input but also a hidden state computed from last timestep to produce a prediction and a new hidden state. The ability to keep an internal memory of previous information makes RNN suitable for processing sequences of data where previous data in the sequence influences subsequent data. For example, when translating speech between languages the word “she” may refer to a person discussed in a previous sentence. Mathematically, Vanilla RNNs are expressed as shown in Figure 3 and as follows in Equation (1):

$$h_t = \sigma(W_{hh}h_{t-1} + W_{xh}i_t + B_h) \qquad (1)$$

where W_{hh} and W_{xh} represent the recurrent weight matrix, i_t represents the local input for each timestep, and h_t represents hidden state output. These components often uses matrix operations such as GEMMs to perform their computations. The computations are tiled into blocks and threads and run in parallel.

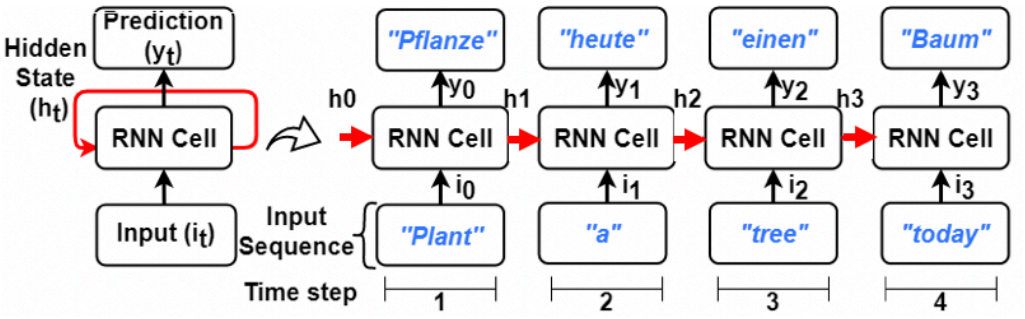

Figure 2: The sequential dependencies in RNN cause inefficiencies. While the computations involving input i_t can be conducted in parallel, the dependencies on h_t cannot be eliminated. Figure reproduced with permission from Pati, et al. [16].

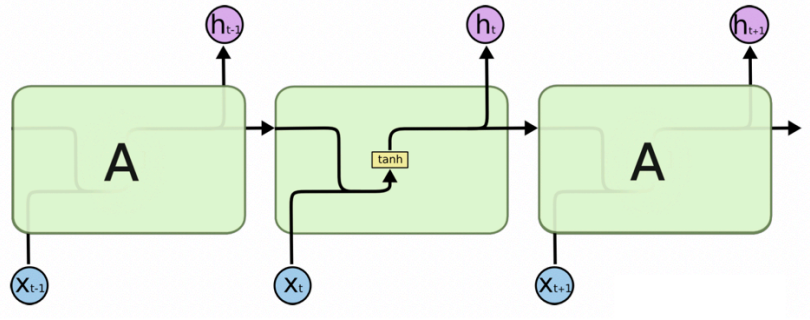

Figure 3: The computations in a single RNN Vanilla cell. Figure adapted from Olah [17].

However, the data dependencies on the hidden states from previous timesteps (h_{t-1}) make it hard to exploit parallelism on modern GPUs, even though RNNs use GEMMS which are often efficient on GPUs. As a result, the most common RNN implementations launch hundreds of GPU kernels, each of which roughly correspond to a single timestep. In this way, RNNs use the kernel boundaries to implicitly synchronize between threads and guarantee that the hidden states are ready before next timestep begins. However, kernel launches, copying data back and forth between CPU and GPU, and moving data into the GPU's fast, local registers and shared memory are time-consuming, and therefore cause inefficiencies.

2.2 Persistency

Previous work on applying persistence to GPU RNNs utilizes the insight that the computations in each timestep of RNN use the same recurrent weight matrix (W_{hh}, W_{xh} in Equation (1)) [1]. Since the weight matrix can be reused over multiple timesteps, it is inefficient to repeatedly load it from global memory once per kernel. To avoid the reloading, persistent computation kernels were developed by exploiting kernel fusion and the reuse of recurrent weight matrix. It also uses shared memory to store hidden states loaded from global memory for use by multiple threads to do the computation in parallel. In PRNN, the recurrent weight matrix is divided into contiguous blocks of rows, stored in registers, and reused over multiple timesteps. The computations involving the

recurrent weight matrix and the inputs ($W_{xh}i_t$) can thus be parallelized by combining multiple inputs as a single large matrix and performing matrix multiplication on it. This combination partly alleviates the lack of parallelism caused by sequential dependencies on hidden states from previous timesteps. As a result, PRNNs have been shown to outperform regular RNNs [1]. As shown in Figure 4, PRNN has better throughput than regular GEMM-based RNN implementations especially at small mini-batch sizes. It also has better throughput scaling with the increasing of timesteps and number of GPUs.

2.3 GRU

Although Vanilla RNNs are simple, they are often unable to retain long-term dependence information. As a result, their accuracy often suffers and more complex RNN variants such as GRU were introduced. As shown in Figure 5 GRU adds additional sub-computations in each timestep. The extra computations help GRU decide which longer-term information is more relevant and should be kept, and which information is less important and can be discarded. As a result, GRU makes more precise predictions than Vanilla RNNs on certain tasks, and is thus more widely applied in real world. Mathematically, GRU RNNs are expressed as follows in Equations (2) – (5):

$$z_t = \sigma(W_{zh}h_{t-1}+W_{zx}x_t+B_z) \quad (2)$$

$$r_t = \sigma(W_{rh}h_{t-1}+W_{rx}x_t+B_r) \quad (3)$$

$$\tilde{h}_t = \tanh(x_tW_{xh}+(\mathbf{r_t}\odot\mathbf{h_{t-1}})W_{hh}+B_h) \quad (4)$$

$$h_t=\mathbf{z_t}\odot\mathbf{h_{t-1}}+(1-\mathbf{z_t})\odot\tilde{\mathbf{h}}_\mathbf{t} \quad (5)$$

where W_{zh} and W_{zx}, W_{rh} and W_{rx}, W_{xh} and W_{hh} are recurrent weight matrices. For simplicity, W_{zh} and W_{zx} are defined to be the same format as the Baidu implementation [4], as well as W_{rh} and W_{rx}, and W_{xh} and W_{hh}. This will not affect accuracy because weight matrices are all defined randomly at the beginning. x_t represents local input, h_t represents hidden state, z_t represents the update gate which decides how much of the past information needs to be passed along to the future, and r_t represents the reset gate which decides how much of the past information needs to be forgot.

Although GRU improves accuracy compared to Vanilla RNNs, its additional sub-computations also have more data dependencies. The computations in each GRU timestep not only depend on the hidden state (h_{t-1}) from a previous timestep, but also have intra-timestep data dependencies. For example, Equation (4) depends on r_t, the output of Equation (3) and Equation (5) depends on z_t, the output of Equation (2). Thus, to respect these additional, intra-cell dependencies, GRU also requires extra global barriers, which makes reducing synchronization overhead even more important.

3 Understanding PRNN Implementation

We next discuss how Baidu's open-source Vanilla PRNN algorithm [4] works.

3.1 Math in PRNN

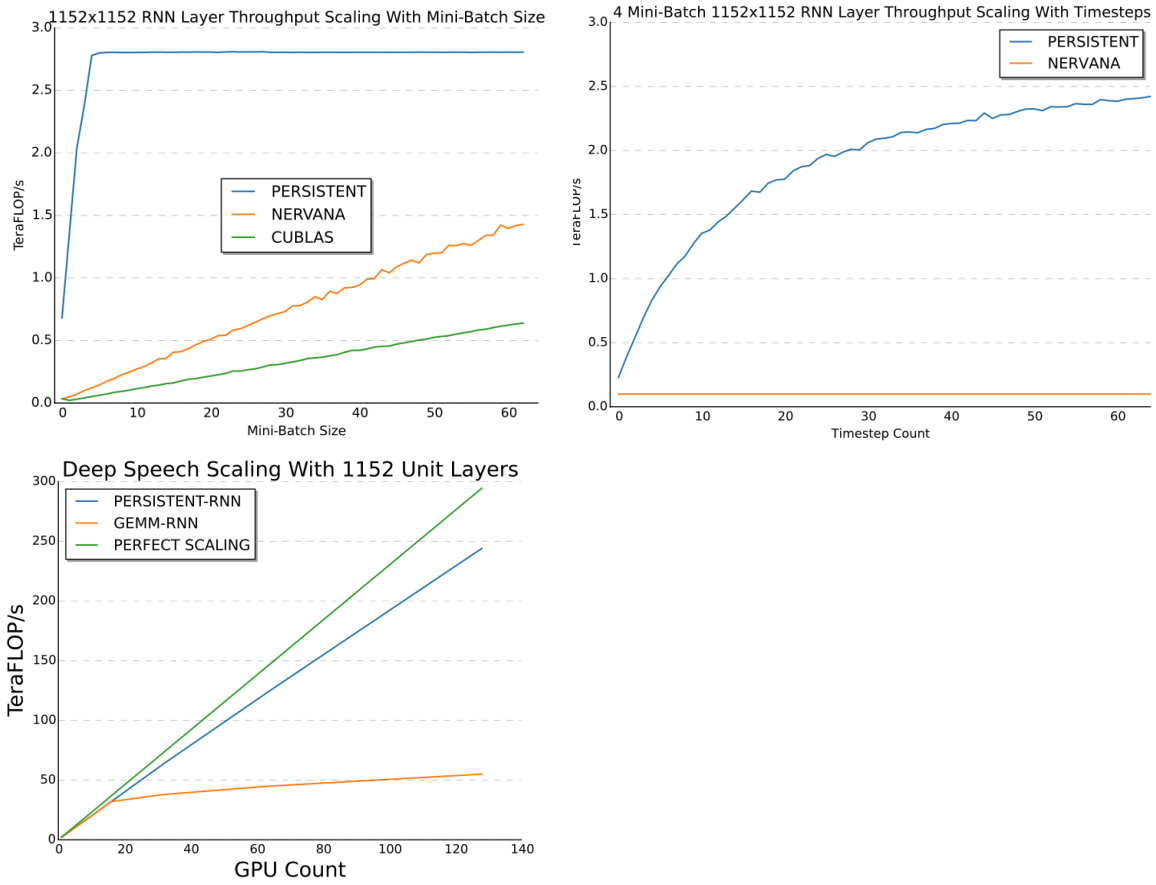

Figure 4: Previous work [4] has shown that PRNNs give better throughput at smaller mini-batch sizes and better scaling as timesteps and GPU counts increase.

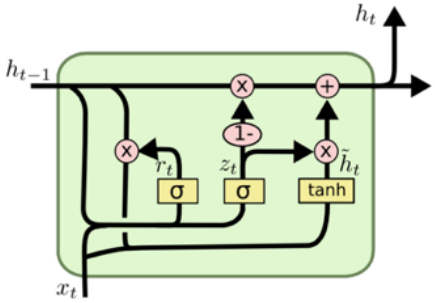

Figure 5: The computations in a single RNN GRU cell. Figure adapted from Olah [17].

The Vanilla PRNN implementation stores its recurrent weight matrix in local registers so that it can be reused across multiple timesteps. However, since the register file in GPU is limited in size, we must carefully choose the hidden layer size to ensure that the recurrent weight matrix fits in local registers. The hidden layer size must satisfy the following Equation (6):

$$3 \times (4 \text{ Bytes} \times s^2) \leq r_s \quad (6)$$

where s is the hidden layer size (number of floats) and r_s is the size of register file. Equation (6) also includes 3 and 4 $Bytes$ because GRU has three recurrent weight matrices and the size of float is 4 bytes. For example, on a Pascal-class GTX 1080, r_s is 20 MB; thus s should be no larger than 1321 floats. We can also obtain the max hidden layer size for newer NVIDIA GPUs using the Equation (6).

3.2 The computations of $W_{xh}x_t$

As discussed in Section 2.2, the computation of $W_{xh}x_t$ in Equation (1) has no dependency on previous timesteps and thus can be done in parallel, off the critical path. Through examining the Baidu's PRNN implementation [4], we found that they used random numbers to represent the computations of $W_{xh}x_t$ without doing the real computations. This approach does not affect the evaluation of PRNN performance because the computations of $W_{xh}x_t$ can be done off the critical path. However, because of this shortcut, the Vanilla PRNN implementation cannot be applied directly to real-world tasks.

3.3 Pipeline

As shown in Figure 6, the PRNN implementation utilizes a three-stage pipeline.[1] As a result, the PRNN pipeline can execute different parts of three timesteps (*t-1*, *t*, *t+1*) in parallel in each iteration *i* and hide the latency of memory operations with compute operations. For example, in iteration *i*, the first stage checks for barrier success and loads hidden states from timestep *t-2* into shared memory for use in timestep *t+1*. In the second stage, it performs the matrix multiplication of timestep t and stores the results into shared memory. Finally in the last stage, it gets the hidden state of timestep *t-1* by reducing the results computed by multiple threads and stores the output into global memory.

3.4 Data buffer and data mapping

Since PRNN uses a pipeline and executes different parts of multiple timesteps in each iteration, it needs to buffer data from a given timestep in shared memory until it is used in a later iteration. Continuing with the example in Section 3.3, in iteration *i*, hidden state outputs of timestep *t-2* are loaded into shared memory for use by timestep *t+1* in the next iteration. Then in iteration *i+1*, hidden state outputs of timestep *t-1*will also be loaded into shared memory while the hidden state outputs of timestep *t-2* will be used by math operations. Therefore, we need to buffer two sets of hidden states in the shared memory at the same time. To achieve this, PRNN uses a two-slot ping-pong buffer (*shared_state* in Baidu's implementation [4]) so that in each iteration, the load operation writes data to half of the ping-pong buffer while the math operations read from the other half. The size of the ping-pong buffer is determined by the grid tile size, block tile size, and thread tile size, which are configurations of the kernels and can be set by hand. PRNN also uses a data structure (called *RegisterState* in Baidu's implementation [4]) to keep track of the starting point of data buffers and help compute the access offset.

[1] Baidu's initial pipeline implementation had initially four stages to mirror the ratio of latency of the load operation and the math operations in one timestep [1]. However, it was later updated to a three-stage pipeline; we believe this change was made to adapt to changes in the latency ratio of memory and math operations.

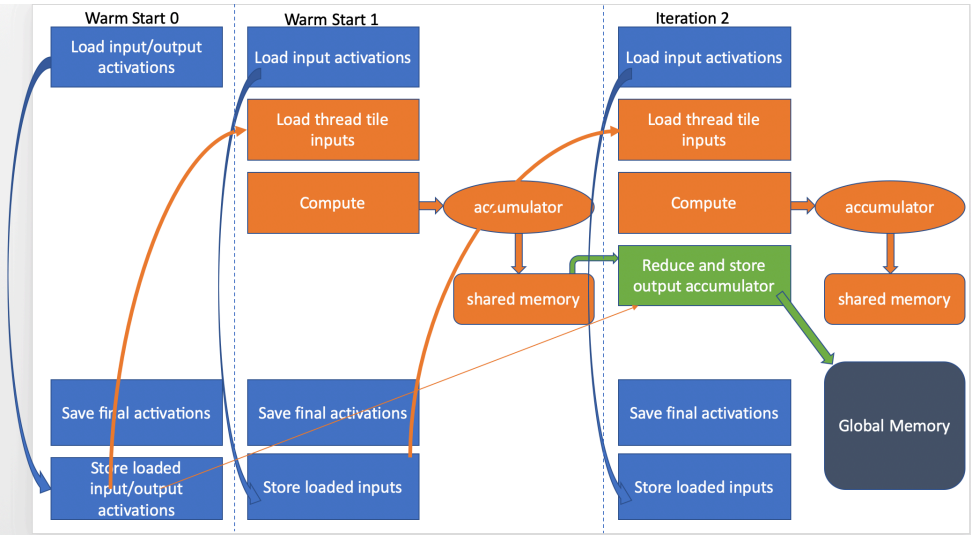

Figure 6: The three-stage pipeline of PRNN.

3.5 Verifying outputs with Python model

To ensure that the PRNN implementation works correctly, we implemented a Vanilla RNN model in Python. When giving the same random inputs, the PRNN model and the Python model give the same outputs.

4 Implementation of Persistent GRU

4.1 Designing the pipeline for PGRU

GRU has more sub-computations than a Vanilla RNN. Also, in each GRU cell, there are data dependencies between the sub-computations that must be respected. Accordingly, to extend the Vanilla PRNN model to work for GRU, we developed a PGRU pipeline that includes all required computations and data dependencies. As shown in Figure 7, in this pipeline the first three stages correspond to GRU Equations (2) and (3) and generate r_t and z_t. Stages four through six correspond to GRU Equation (4) and compute of $\tilde{h}_t$. Finally, the last two stages compute the final hidden state h_t.

4.2 Design data structures for PGRU

Since each iteration in the PGRU pipeline is executing parts of multiple timesteps, some intermediary results and data inputs must be kept in shared memory for multiple iterations until they can be used (similar to the hidden states in PRNN in Section 3.4). For example, in the PGRU pipeline, the h_{prev} (h_{t-1} in Equations (2)-(5)) loaded in iteration 1 is still used in iteration 5 by the computation of $h_{candidate}$ ($\tilde{h}_t$ in Equation (4)(5)). Similarly r_t and z_t are generated in iteration 5; although r_t is immediately used in iteration 6, z_t is used later in iteration 7. So, to keep these required data in shared memory, we extend the ping-pong buffer from PRNN implementation, but expand the size of the buffer to store more than two sets of data since some data must be kept longer than two iterations in PGRU. For the same example of h_{prev}, we need a five-slot buffer to keep it in the shared memory for five iterations.

However, having more slots in the data buffers and more such large-sized buffers may demand too much shared memory per thread block (TB). This is a limitation of our approach, as it requires a smaller data buffer size than Vanilla PRNNs (accomplished by reducing the grid tile size, block tile size, and thread tile size). To enable better scalability for different sized tasks, future PGRU designs may need to fundamentally rethink the structure of data buffers in the future.

4.3 Implement the first three iterations of PGRU pipeline

The first three iterations of the PGRU pipeline compute z_t and r_t from Equations (2) and (3). These equations have the same structure as Equation (1) in Vanilla RNNs. As a result, we need to double the computations and data structures for Vanilla RNN. The main tasks are 1) correctly sending required data to kernels and 2) correctly mapping data in the memory by keeping track of the starting addresses and computing access offset properly.

We handle the first task by creating two recurrent weight matrices, one each for Equation (2) and (3), and two sets of random numbers as placeholders for the two computations involving local inputs ($W_{zx}x_t$ in Equation (2) and $W_{rx}x_t$ in Equation (3)). Then we modify the signatures of all the necessary functions that send these new data into the kernels.

The second task is more complex. Besides doubling the data buffer (*shared_state* in Baidu's implementation [4]) and doubling the state-tracking structure (*RegisterState* in Baidu's implementation [4]), we need to carefully choose the sizes of grid tiles, block tiles, and thread tiles to avoid using too much shared memory. By examining the definition of buffer size, we found that the buffer size depends on the number of block tile rows, block tile columns, and thread tile columns. Thus, to allow PGRU to fit in the available registers and shared memory (after setting the CUDA flag to prefer shared memory in the kernels), we set the number of grid tile columns and grid tile rows to 384, the number of block tile rows to 128 and block tile columns to 256, and the number of thread tile rows to 6 and thread tile columns to 36.

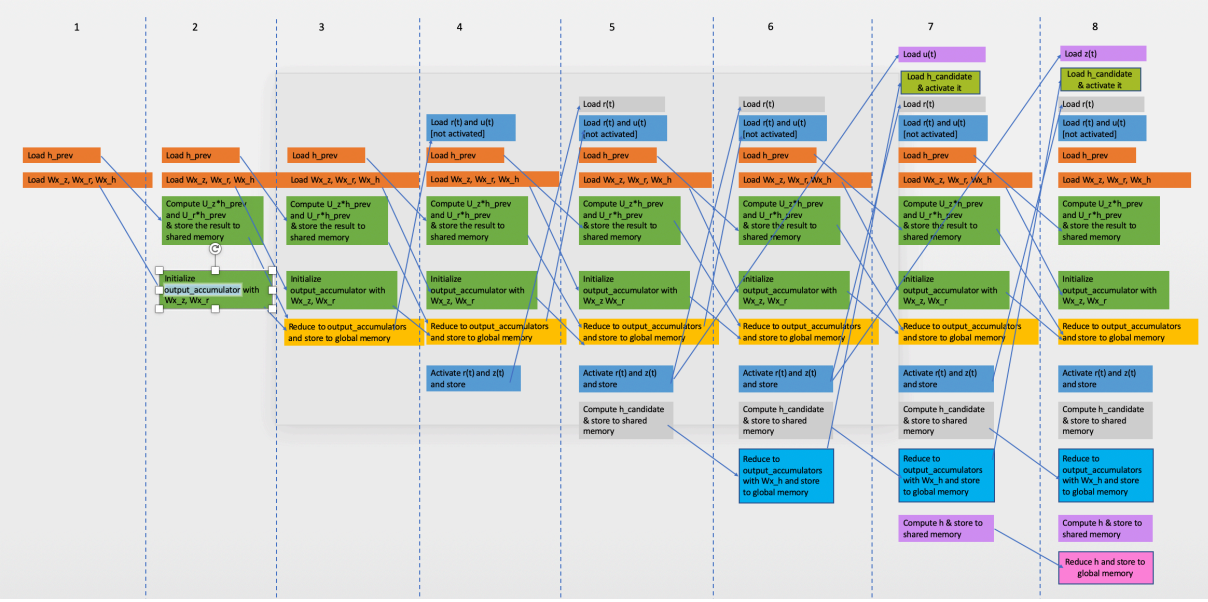

Figure 7: Proposed pipeline for PGRU.

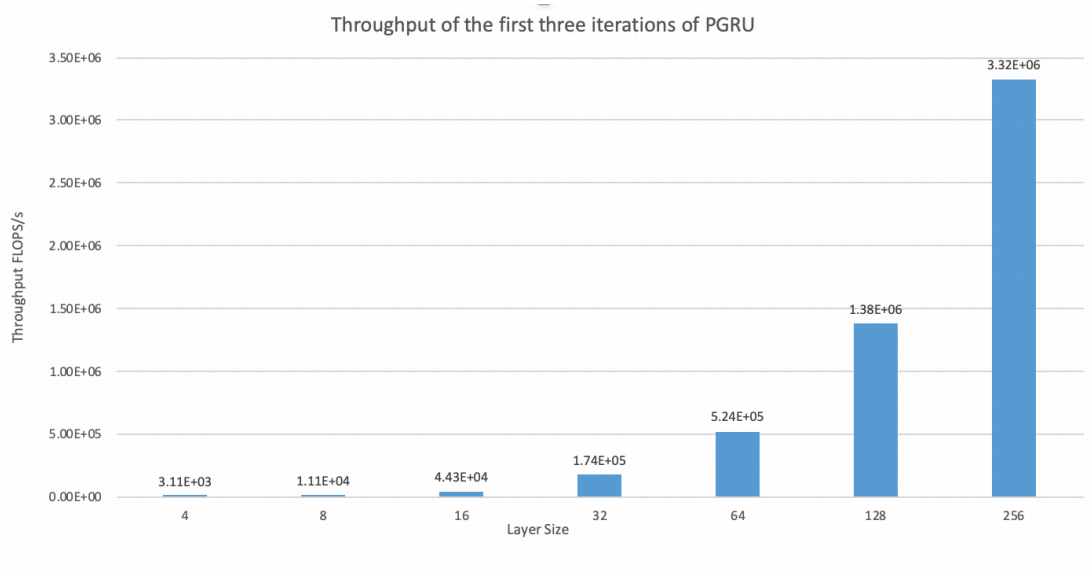

Figure 8: The throughput performance of the first three iterations in PGRU, as hidden layer size varies.

5 Results

5.1 Methodology

We compiled and ran the implementation of the first three iterations of PGRU on a Pascal GTX 1080 GPU, which has 20 SMs, 256 KB of register file per SM, and 96 KB of shared memory per SM. For our experiments, we use CUDA 11.2. Since these experiments are intended as a proof of concept, we use a very small network to test our implementation: 1 iterations, 1 RNN layers, 4 timesteps, and a mini-batch size of 4. We then ran the experiments with hidden layer size 2^n with n ranging from 2 to 8. To ensure that our PGRU implementation was giving correct results, we extend our Python model (Section 3.5) to support PGRU and verified all outputs matched the output of the Python model.

5.2 Results of the first three iterations of PGRU

Figure 8 shows the throughput of the first three stages of the PGRU pipeline as hidden layer size varies. It displays a trend of the throughput increasing with increasing hidden layer size. This is expected because as larger layer size increases, more computations are executed in parallel and therefore throughput increases.

5 Future work

Thus far we have finished implementing the first three iterations of the PGRU pipeline. Next we will finish implementing PGRU by adding the later iterations in the pipeline. After that, we will utilize our PGRU implementation to reduce synchronization overhead in PRNNs. Our key insight is that PRNN synchronization is too coarse. For example, Figure 9a shows the computations in one GRU cell. Some of these computation (e.g., 1-$u(t)$), do not depend on others (e.g., $r(t)$). Similarly, $z(t)$ does not depend on $u(t)$. Thus, having a global barrier across all of these

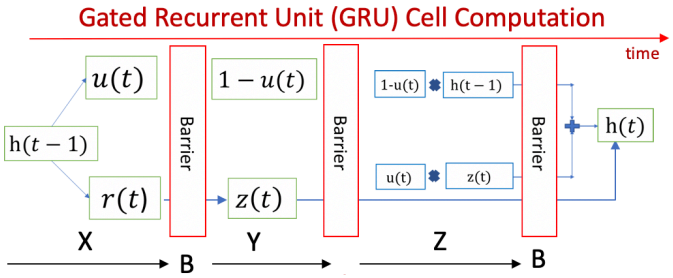

Figure 9a: GRU cell with global barriers.

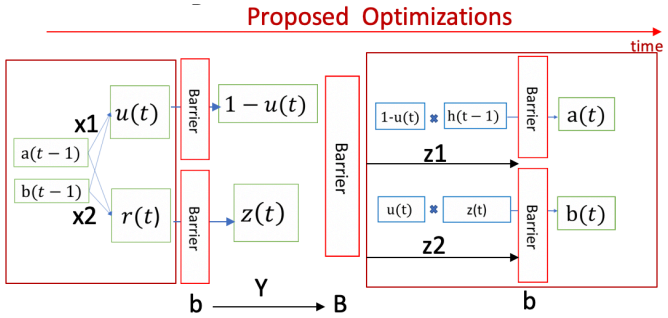

Figure 9b: GRU cell with local barriers.

computations is not necessary. Instead, if we co-locate these independent partial computations, we only need to use local synchronization. More precisely, if we can map independent partial computations onto one single Streaming Multiprocessor (SM), the synchronization can be performed at the local L1 cache, which is much faster to access than the L2 cache. Even if we can only map independent partial computations onto a small number of SMs, we can still significantly reduce the contention of the synchronization variables.

Based on this understanding, what we will do next is at first dividing GRU into independent partial computations. Based on GRU's algorithmic properties, we can easily identify computations that are not dependent on all previous calculated variables. Next, we map these independent computations onto designated SMs. To do so, we will leverage the round-robin scheduler in NVIDIA GPUs to assign computations to specific SMs. Once we have independent computations on specific SMs, we can replace global barriers with local barriers for the threads running on those SMs. As a result, this approach will improve PRNN performance by increasing parallel computations and better overlapping memory accesses. Analytically, as shown in Figure 9, when using global barriers, the execution time of one GRU cell is: $X + B + Y + B + Z + B$. After dividing GRU into independent partial computations and using local barriers, the execution time becomes: $\max(x1, x2) + b + Y + B + \max(z1, z2) + b$, which is faster by at least $2(B\text{-}b)$.

6 Conclusion

RNNs are an important category of neural network models that are wildly used in real applications. However, its common implementations on GPU are not very efficient due to the large number of kernel launching and large amount of data transferring. PRNN seeks to improves the performance of RNN on GPU by kernel fusion and reuse of recurrent weight matrix. By fusing the large number of kernels into a single one, and storing the recurrent weight matrix locally in registers, PRNN

saves the time of kernel launching, data transferring, and repeatedly loading data from the main memory. Moreover, PRNN increases parallelism by performing independent computations in parallel. However, it increases the use of global synchronization. To expand the benefits of persistency and better investigate its synchronization overhead for more popular RNN variants, we design a persistent GRU implementation. Thus far we have examined the PRNN implementation, developed the pipeline for PRNN and PGRU, and finished the implementation of the first three stages of PGRU. We have also identified a series of potential opportunities to improve the performance of PGRU such as further simplifying the pipeline and designing more efficient data buffers.

Acknowledgements

This work was generously supported by the University of Wisconsin-Madison L&S Honors Program through a Trewartha Senior Thesis Research Grant awarded by the L&S Honors Program. The author would also like to whole-heartedly thank Prof. Matt Sinclair and Preyesh Dalmia for their inspiring guidance and patient, repeated help along the way.

References

[1] Gregory Diamos, Shubho Sengupta, Bryan Catanzaro, Mike Chrzanowski, Adam Coates, Erich Elsen, Jesse Engel, Awni Hannun, and Sanjeev Satheesh. 2016. Persistent RNNs: stashing recurrent weights on-chip. In Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48 (ICML'16), 2024–2033.

[2] M. D. Sinclair, J. Alsop and S. V. Adve, "HeteroSync: A benchmark suite for fine-grained synchronization on tightly coupled GPUs," 2017 IEEE International Symposium on Workload Characterization (IISWC), 2017, pp. 239-249, doi: 10.1109/IISWC.2017.8167781.

[3] J. A. Stuart and J. D. Owens, "Efficient Synchronization Primitives for GPUs," CoRR, vol. abs/1110.4623, 2011. [Online]. Available: http://arxiv.org/abs/1110.4623

[4] Baidu Research, Persistent RNN Github, https://github.com/baidu-research/persistent-rnn.

[5] Dario Amodei, Sundaram Ananthanarayanan, Rishita Anubhai, Jingliang Bai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Qiang Cheng, Guoliang Chen, Jie Chen, Jingdong Chen, Zhijie Chen, Mike Chrzanowski, Adam Coates, Greg Diamos, Ke Ding, Niandong Du, Erich Elsen, Jesse Engel, Weiwei Fang, Linxi Fan, Christopher Fougner, Liang Gao, Caixia Gong, Awni Hannun, Tony Han, Lappi Vaino Johannes, Bing Jiang, Cai Ju, Billy Jun, Patrick LeGresley, Libby Lin, Junjie Liu, Yang Liu, Weigao Li, Xiangang Li, Dongpeng Ma, Sharan Narang, Andrew Ng, Sherjil Ozair, Yiping Peng, Ryan Prenger, Sheng Qian, Zongfeng Quan, Jonathan Raiman, Vinay Rao, Sanjeev Satheesh, David Seetapun, Shubho Sengupta, Kavya Srinet, Anuroop Sriram, Haiyuan Tang, Liliang Tang, Chong Wang, Jidong Wang, Kaifu Wang, Yi Wang, Zhijian Wang, Zhiqian Wang, Shuang Wu, Likai Wei, Bo Xiao, Wen Xie, Yan Xie, Dani Yogatama, Bin Yuan, Jun Zhan, and Zhenyao Zhu. 2016. Deep speech 2: end-to-end speech recognition in English and mandarin. In Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48 (ICML'16), 173–182.

[6] S. Xiao and W. Feng, "Inter-block GPU communication via fast barrier synchronization," 2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS), 2010, pp. 1-12, doi: 10.1109/IPDPS.2010.5470477.

[7] Hochreiter S, Schmidhuber J. Long short-term memory. Neural Computation, 9(8): 1735-1780, 1997.

[8] Cho K, Bahdanau D, Bougares F, Schwenk H and Bengio Y. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP 2014), 2014.

[9] Chung J, Gulcehre C, Cho K, Bengio Y. Empirical evaluation of gated recurrent neural networks on sequence modelling. arXiv preprint arXiv:1412.3555, 2014.

[10] Tomáš Mikolov. 2012. Statistical language models based on neural networks. Presentation at Google, Mountain View, 2nd April (2012).

[11] Kai Sheng Tai, Richard Socher, and Christopher D. Manning. 2015. Improved Semantic Representations From Tree-Structured Long Short-Term Memory Net- works. In Proc. ACL, Volume 1: Long Papers. 1556–1566.

[12] Chun-Chi J. Chen and Risto Miikkulainen. Creating Melodies with Evolving Recurrent Neural Networks. Proceedings of the 2001 International Joint Conference on Neural Networks, 2001.

[13] Douglas Eck and Jurgen Schmidhuber. A First Look at Music Composition Using LSTM Recurrent Neural Networks. Technical Report No. IDSIA-07-02, 2002.

[14] R. Adolf, S. Rama, B. Reagen, G. Wei and D. Brooks, "Fathom: reference workloads for modern deep learning methods," 2016 IEEE International Symposium on Workload Characterization (IISWC), 2016, pp. 1-10, doi: 10.1109/IISWC.2016.7581275.

[15] Nvidia cuDNN Documentation Developer Guide 10.3 https://docs.nvidia.com/deeplearning/cudnn/developer-guide/

[16] Suchita Pati, Shaizeen Aga, Matthew D. Sinclair, and Nuwan Jayasena. "SeqPoint: Identifying Representative Iterations of Sequence-based Neural Networks." In 2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pp. 69-80. IEEE, 2020.

[17] C. Olah, Understanding LSTM Networks, 2015. http://colah.github.io/posts/2015-08-Understanding-LSTMs/.