Persistent RNNs: Stashing Recurrent Weights On-Chip

Gregory Diamos   GREGDIAMOS@BAIDU.COM
Shubho Sengupta   SSENGUPTA@BAIDU.COM
Bryan Catanzaro   BCATANZARO@BAIDU.COM
Mike Chrzanowski   MIKECHRZANOWSKI@BAIDU.COM
Adam Coates   ADAMCOATES@BAIDU.COM
Erich Elsen   ERICHELSEN@BAIDU.COM
Jesse Engel   JENGEL@BAIDU.COM
Awni Hannun   AWNIHANNUN@BAIDU.COM
Sanjeev Satheesh   SANJEEVSATHEESTH@BAIDU.COM

Baidu Silicon Valley AI Lab, 1195 Bordeaux Drive, Sunnyvale, CA 94089, UNITED STATES

Abstract

This paper introduces a new technique for mapping Deep Recurrent Neural Networks (RNN) efficiently onto GPUs. We show how it is possible to achieve substantially higher computational throughput at low mini-batch sizes than direct implementations of RNNs based on matrix multiplications. The key to our approach is the use of persistent computational kernels that exploit the GPU's inverted memory hierarchy to reuse network weights over multiple timesteps. Our initial implementation sustains 2.8 TFLOP/s at a mini-batch size of 4 on an NVIDIA TitanX GPU. This provides a 16x reduction in activation memory footprint, enables model training with 12x more parameters on the same hardware, allows us to strongly scale RNN training to 128 GPUs, and allows us to efficiently explore end-to-end speech recognition models with over 100 layers.

1. Introduction

Recurrent Neural Networks (RNNs) have been shown to be powerful tools for solving general sequence to sequence mapping problems in domains ranging from speech recognition (Sainath et al., 2015) to natural language processing (Gao et al., 2015)(Sutskever et al., 2014).

In this paper, we explore techniques for mapping RNNs to throughput optimized processors such as GPUs. We use the Multi-Bulk-Synchronous-Parallel (Valiant, 2008) (MBSP)

model to analyze the computation, communication, and synchronization operations performed by a RNN. We focus on mapping strategies that carefully manage data movement through the processor's memory hierarchy to balance these costs. These changes enable RNN implementations on GPUs that are very efficient at small mini-batch sizes, even on mini-batch sizes of just 4 examples. We exploit this reduction in batch size to decrease the memory footprint of our networks by 16x, allowing us to explore deeper networks without exceeding GPU memory.

We exploit the largest source of on-chip memory on the GPU—the collective register files of 6144 hardware thread contexts on a TitanX GPU—to cache the RNN parameters and reuse them over multiple timesteps during training. We attack the cost of inter-processor synchronization with an optimized assembly level barrier implementation, demonstrating that such barriers implemented in software can reduce latency by approximately 10x compared to relying on repeated kernel launches.

To make our results relevant for deployment, we only consider models with a hard constraint of 800ms of future context. We find that accuracy improves with deeper models using batch normalization and skip connections (He et al., 2015; Srivastava et al., 2015), reinforcing the trend towards deeper models in vision applications. We present evidence that accuracy continues to improve with increased depth.

2. Related Work

This work is motivated by recent advances in speech recognition and natural language processing using deep RNNs. It draws insight from related work on performance optimization of DNN and dense linear algebra libraries, distributed training of DNNs, work on general purpose GPU

performance optimization, and high performance processor technology trends.

In computer vision recognition tasks, Deep Neural Nets (DNN) have demonstrated superior performance in object detection (Krizhevsky et al., 2012; Simonyan & Zisserman, 2014; Szegedy et al., 2014; He et al., 2015), localization (He et al., 2015), and pose estimation (Toshev & Szegedy, 2013). In natural language processing, DNNs have enabled significant advancements in language modeling (Bengio et al., 2003; Mikolov et al., 2010), sentiment analysis (Socher et al., 2013; Iyyer et al., 2015; Le & Zuidema, 2015), syntactic parsing (Collobert & Weston, 2008; Socher et al., 2011; Chen & Manning, 2014) and machine translation (Bahdanau et al., 2014; Devlin et al., 2014; Sutskever et al., 2014).

In speech recognition, DNNs have become a fixture in the ASR pipeline (Mohamed et al., 2011; Hinton et al., 2012; Dahl et al., 2011b;a; N. Jaitly & Vanhoucke, 2012; Seide et al., 2011). CNNs have also been found beneficial for acoustic models (Abdel-Hamid et al., 2012; Sainath et al., 2013). RNNs, typically LSTMs, arecommonly used in state-of-the art recognizers (Graves et al., 2013; H. Sak et al., 2014; Sak et al., 2014) and work well together with convolutional layers for feature extraction (Sainath et al., 2015). End-to-end speech recognition with a combination of CNNs and RNNs has also been developed (Amodei et al., 2015).

As DNNs have continued to increase application-level performance, more effort has been applied to hardware and software optimizations targeting DNNs. High performance libraries following a similar design philosophy as BLAS have emerged (Chetlur et al.). These libraries have begun to include optimized RNN routines, although they have not yet used persistent GPU kernels. Additional work has focused on improving algorithmic efficiency of the fundamental operations used by DNNs (Vasilache et al., 2014; Lavin & Gray, 2015). Finally, DNNs often rely on dense linear algebra operations, and benefit from prior work that has resulted in highly tuned implementations for modern processors (Dongarra et al., 2014; Gray, 2014).

An open-source implementation of the Persistent RNN GPU kernels has been released (Diamos et al.).

3. RNN to Hardware Mapping Strategy

Let a single input sequence x and corresponding output sequence y be sampled from a training set $\mathcal{X} = \{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots\}$. Each input sequence, $x^{(i)}$, is a time-series of length $T^{(i)}$ where every time-slice is a vector of application-specific features (e.g. audio samples), $x_t^{(i)}, t = 0, \ldots, T^{(i)-1}$.

The forward in time h^l recurrent layer activations are

$$h_t^l = f(h_t^{l-1}, h_{t-1}^l) \tag{1}$$

The function f can be the standard recurrent operation

$$h_t^l = \sigma(W^l h_t^{l-1} + U^l h_{t-1}^l + b^l) \tag{2}$$

where W^l is the input-hidden weight matrix, U^l is the recurrent weight matrix and b^l is a bias term.

Implementations of recurrent neural networks typically separate the computation into two stages.

In the first stage ($W^l h_t^{l-1}$), the contribution to the output of each neuron for each timestep is computed using the neuron inputs for that timestep. Like a feed forward network, the first stage represents the input weights of all the neurons in the layer as a dense two-dimensional matrix and the inputs to the layer for each timestep as a one-dimensional dense vector. A common optimization is to unroll the time dimension and pack multiple one-dimensional input vectors together into a single two-dimensional matrix. This is possible because the weight matrix is shared over all timesteps.

In the second stage ($U^l h_{t-1}^l$), the connections between the outputs of the layer neurons on a given timestep to the inputs of the layer neurons on the next timestep are represented by a two-dimensional matrix, referred to as the recurrent weight matrix. In this case, each timestep must be processed sequentially because the outputs of the next timestep depend on the outputs of the current timestep, requiring this operation to be performed using a matrix-vector product, followed by an application of the activation function. This is the most computationally expensive step, since the sequential dependence between timesteps requires explicit synchronization between them and the recurrent weight matrix has to be reloaded from memory on each timestep. So we focus our attention on optimizing this stage using the Multi-Bulk-Synchrononus-Parallel machine model.

3.1. Multi-Bulk-Synchronous-Parallel Machine Model

The Multi-Bulk-Synchronous-Parallel (MBSP) abstract machine model (Valiant, 2008) is a processor performance model that takes into account the physical realities of multiple processor cores, each with finite memory and computational resources, as well as communication and synchronization costs. These costs are a function of the physical dimensions of and distances between processor cores. They typically increase with the number of cores.

The MBSP model is a hierarchical model, with an arbitrary number of levels. At each level, it describes a collection of processor cores and the associated on-chip memory/cache capacities with four parameters (computational bandwidth,
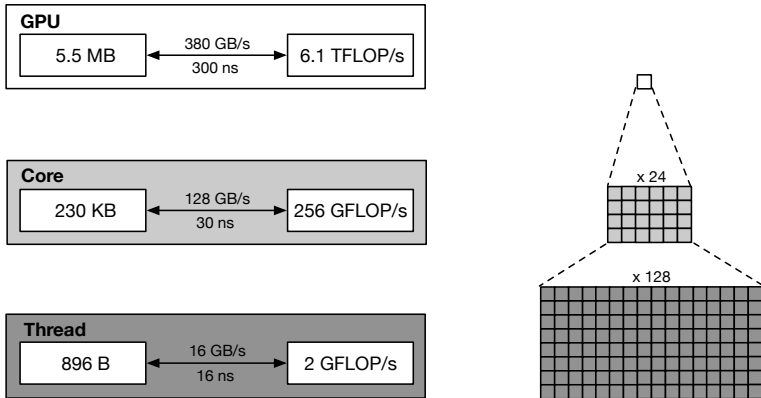
Figure 1: A depiction of the MBSP machine model hierarchy for an NVIDIA TitanX GPU. Each level of the hierarchy contains memory capacity (left square) with associated latency and bandwidth (arrows), as well as a set of sublevels or compute units (right square). At each level of the hierarchy, the Persistent RNN kernels store all neuron parameters in memory and match latencies between communication, synchronization, and computation.

memory capacity, memory bandwidth, and memory latency). A memory hierarchy is needed due to the physical limits on the amount of memory that can be accessed in a fixed amount of time from a processor core.

We first generate the MBSP model for the target processor or family of processors that we plan to execute our RNN. Our model describes each level of a processor's memory hierarchy with a tuple (p, b, c, m), where p represents the number of submodules or processor cores, b represents the communication bandwidth, c represents the synchronization cost among processors at this level, and m represents the cache or memory capacity at this level. An example of these parameters for the TitanX GPU is shown in Figure 1.

Starting with the lowest level of hierarchy and working up to the highest, we divide individual neurons into logical modules and arrange the connections between modules such that the following constraints are met:

- the parameters representing the neurons selected for a given processor fit completely into the cache or memory capacity for this level of the memory hierarchy.

- the communication cost implied by intra-module connections and inter-module connections is approximately equal to the computational cost of evaluating the module's neurons.

- the synchronization cost implied by inter-module connections is approximately equal to the computational cost of evaluating the module's neurons.

These changes balance the computational, communication, synchronization, and memory capacity requirements of the RNN such that no one resource becomes a significant bottleneck. It does so by exploiting the reuse of RNN weights

over multiple timesteps to avoid repeatedly loading weights from DRAM, and taking into account the significantly higher cost of synchronization and off-chip memory accesses as compared to floating-point math operations.

4. Implementation on a TitanX GPU

The peak floating point throughput of a TitanX is 6.144 TFLOP/s. A straightforward implementation of a RNN using GEMM operations achieves 0.099 TFLOP/s at a layer size of 1152 using Nervana Systems GEMM kernels at a mini-batch size of 4. Our initial Persistent RNN implementation with the same layer and mini-batch size achieves over 2.8 TFLOP/s resulting in a 30x speedup.

4.1. Approach

In contrast to approaches based on matrix multiplication, we divide the recurrent weight matrix into blocks of contiguous rows, each of which is processed by a single SM , as shown in Figure 2 . Even though this approach requires more global memory bandwidth than the more traditional approach of dividing into tiles, we chose it because it avoids the need to perform an inter-SM reduction to compute the activations for a given block of rows.

Our implementation first loads the weight matrix into registers. Then each SM loads all of the input activations from the previous timestep from global memory to shared memory, computes the dot product for each row, performs the nonlinearity, writes the result for the current timestep, and performs a global barrier with all other SMs. The latency required to perform the load operations is approximately four times higher than the time required to perform the math operations for a single timestep. So we break the
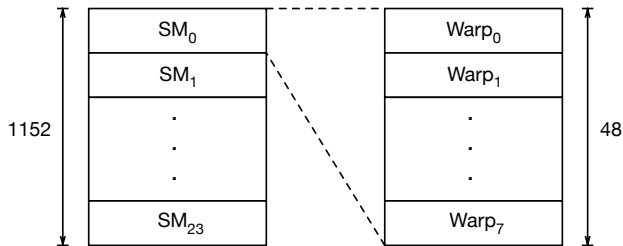
Figure 2: A depiction of the recurrent weight matrix tiling strategy. Each of the 24 SMs on the GPU processes a 48x1152 block row, reading 1152 activations for the current timestep, and writing 48 activations for the next timestep. Within an SM, each of 8 warps processes a 6x1152 block row, and all warps share access to the activations for the current timestep in CUDA shared memory. Groups of 16 threads arranged in an interleaved pattern cooperate to process a 3x1152 tile.

computation into four independent stages and use software pipelining to overlap the load operation with math, reduce and barrier operations. We use a mini-batch size of four or greater to keep the pipeline full.

4.2. Stashing the Weights On-Chip

Each thread in the TitanX GPU has access to approximately 1KB of memory that can be read at high enough bandwidth to saturate the floating point datapath. Out of this, we dedicate 896 bytes to store recurrent weights as shown in Figure 2 , and the rest for intermediate computations. These weights are loaded once at the start of the kernel, and reused over each timestep.

4.3. A Fast Global Barrier

Synchronization between GPU processors cores is typically achieved implicitly between dependent kernel calls in both CUDA and OpenCL development frameworks. However, this mechanism for synchronization between timesteps requires launching a new kernel that forces the weights to be reloaded from off-chip memory. This causes the synchronization latency of dependent kernels to be approximately 6-10x larger than the time spent performing the math operations for a single timestep, and this cannot be overlapped with computation. We address this problem with an optimized implementation of a global barrier that can be completely overlapped with the math operations for a single timestep.

4.4. Saving Memory to Enable Deeper Networks

When training our speech recognition model, we encounter very long utterances that are up to thirty seconds long, cor-

responding to 3,000 timesteps. For a RNN layer with 1760 hidden units, and a mini-batch size of 64, this corresponds to 1.3 GB of storage per layer. This is much more than the 12.3 MB required to store the layer weights. In practice, with GPUs with 12GB of DRAM, we find that this limits us to networks with about 9 layers. A common solution to this problem is to use truncated back-propagation through time (Sutskever, 2013) (BPTT). However, we have observed a 20% relative performance degradation of the converged model using this approach, making other techniques that reduce memory footprint, such as reducing the mini-batch size, more attractive.

4.5. GRUs and LSTMs

The persistent RNN approach can also be applied to GRU and LSTM recurrent networks with simple modifications. In both cases, the update rule for a single timestep can be factored into a component that depends on the layer input h_t^{l-1}, and a component that depends on the output from the previous timestep h_{t-1}^l. As before, the first component can be computed in parallel over all timesteps using matrix multiplications, but the second component requires synchronization between timesteps. Similarly the second component for both GRUs and LSTMs can be factored into a single matrix multiplication by packing the individual recurrent weight matrices together into a single matrix that is applied to the concatenation of each of the activation signals —reset, update, activation, etc. This matrix can now be distributed throughout on-chip memory.

5. Experiments

This section focused mainly on the computational throughput of our Persistent RNN implementation. We also include experimental results of very deep forward-only recurrent networks on a large-scale speech recognition task, similar to the Deep Speech 1 (Hannun et al., 2014) (DS1) and Deep Speech 2 (Amodei et al., 2015) (DS2) systems. These very deep networks would not be feasible to run efficiently on our systems without Persistent RNNs due to memory limitations.

5.1. Computational Throughput

In this section we compare the computational efficiency of Persistent RNN, against an optimized RNN implementation based on matrix multiplication routines from the NVIDIA and Nervana Systems BLAS libraries. We find that our implementation is substantially more efficient at small mini-batch sizes than either of those implementations.
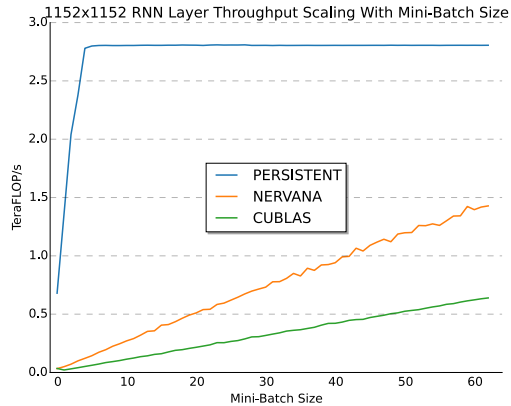
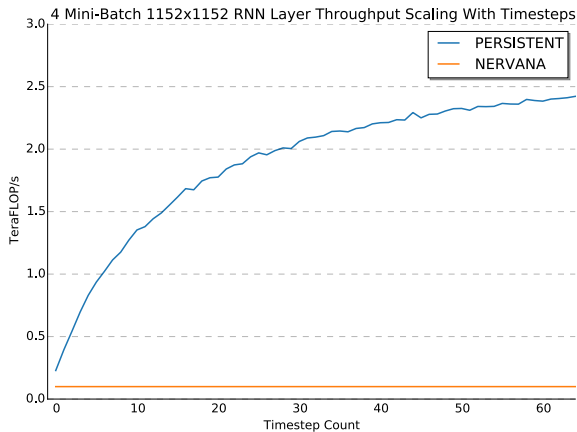Figure 3: Throughput scaling with mini-batch size.

Figure 4: Throughput scaling with timesteps.

5.1.1. SENSITIVITY TO MINI-BATCH-SIZE

Figure 3 compares floating point throughput for Persistent RNN against two other RNN implementations for small mini-batch sizes. Note that after a mini-batch size of four, Persistent RNN consistently deliver approximately 2.8 TFLOP/s, but matrix-multiply based implementations start out much slower, and need relatively large mini batch sizes to become competitive. Even then, we find that layer sizes around 1152 units are somewhat too small for matrix multiplication libraries to be efficient, only achieving about 1.5 TFLOP/s at a mini-batch size of 64. Performance is generally much better at layer sizes of 2560 units, suggesting the advantages of persistent RNN implementations will grow as models become deeper and thinner.

5.1.2. SENSITIVITY TO TIMESTEPS

Figure 4 shows the sensitivity of Persistent RNN to start up overheads associated with launching the kernels and loading the recurrent weight matrix on the first iteration. We
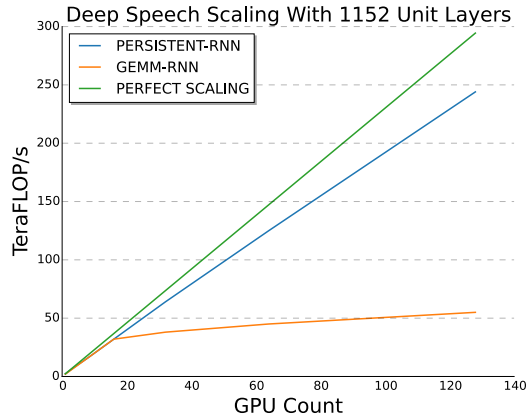

Figure 5: Throughput scaling of the 48 RNN, 61 total layer RNN with a fixed algorithmic mini-batch of 512.

find that most of the performance is achieved after approximately 30 timesteps, which is substantially smaller than the average utterance length in our training set of about 350 timesteps. However, it does suggest that real time implementations that rely on processing a small number of utterances at a time should still buffer up about 30 timesteps (600ms of audio at a frame size of 10ms).

5.1.3. STRONG SCALING

Figure 5 shows scalability of the 48 RNN layer network from 1 to 128 GPUs. Our cluster is composed of nodes with 8 GPUs and 2 CPUs. GPUs are connected locally via PCIe v3 using two 4-wide full bisection bandwidth PCIe switches, which are interconnected using the QPI bus between CPUs. Nodes are interconnected by Infiniband 12x QDR links to a full bisection bandwidth router. We use MPI as the communication layer. We use synchronous SGD as the training algorithm, with data parallelism to support multiple GPUs. There is no need to use a technique that reduces interconnect bandwidth such as asynchronous SGD because our system is fast enough to completely overlap the all-reduce operation in SGD with the back propagation evaluation.

The algorithmic mini-batch size is fixed at 512 for all experiments (i.e. the mini-batch per GPU is 64 when run on 16 GPUs and 4 when run on 128 GPUs). The GEMM based RNN implementation scales well up to 16 GPUs, but does substantially worse as more GPUs are added. The Persistent RNN implementation scales nearly linearly, and achieves 250 TFLOP/s on 128 GPUs (about 30% of peak). Note that this number is the sustained throughput of the entire system, not just the RNN kernels.
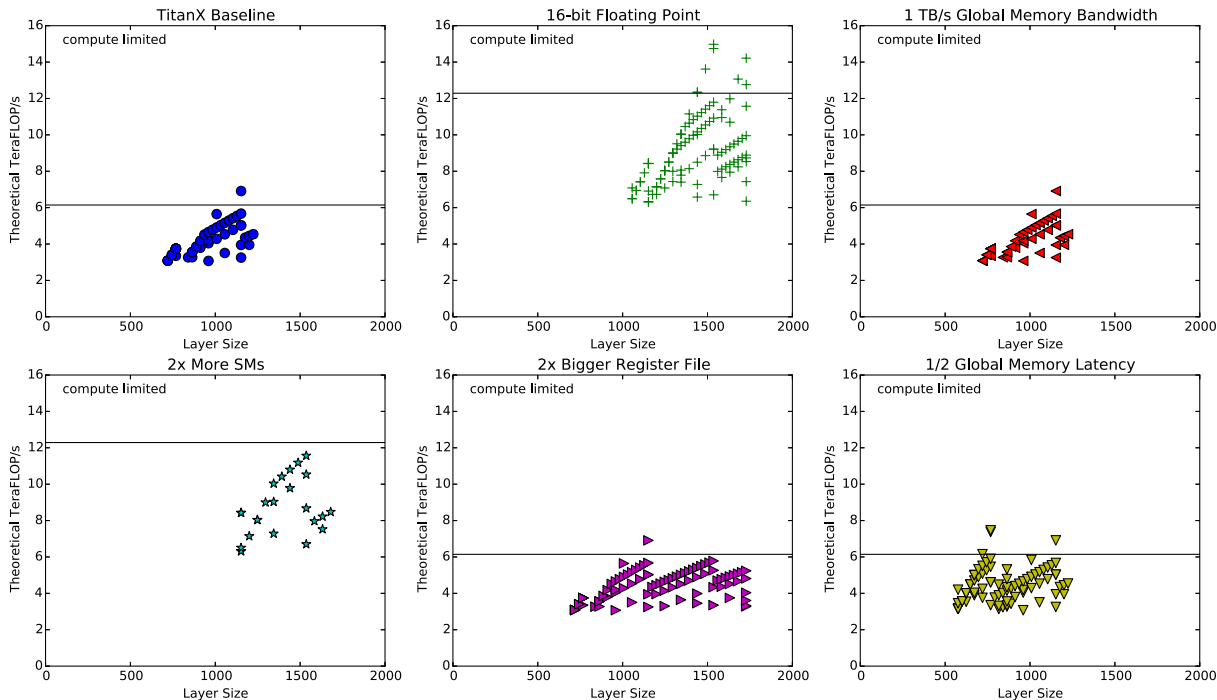
Figure 6: The theoretical floating point throughput for various RNN layer sizes, with different plots for variations of the TitanX GPU microarchitecture. Each point represents a different tiling strategy. The solid line indicates the maximum throughput of that GPU, any point above that line is compute limited, and any point below it is limited by latency, memory bandwidth, or load imbalance.

5.1.4. PROCESSOR DESIGN SPACE

Our final experiment uses an analytical performance model to predict the performance of persistent RNN implementations of various layer sizes on modified versions of the TitanX GPU microarchitecture, offering a view into the performance landscape of future GPUs for different layer sizes. We explore doubling the register file size per SM, doubling the number of SMs, performing all of the computations in 16-bit floating point, doubling the DRAM bandwidth, and halving the memory latency.

Generally, we find that future GPUs will probably enable bigger recurrent layer sizes, and run them at higher throughput. Of the options that are straightforward to implement through process scaling and minor architecture changes, moving to 16-bit floating point operations provides the biggest improvement, followed by more memory per SM, and finally increasing the number of SMs. Reducing synchronization latency is expected to be much more difficult, but it is interesting to note that substantially reducing the memory and synchronization latency would enable the efficient implementation of a larger range of layer sizes (smaller layer sizes in particular). It is also interesting to note that substantially increasing DRAM bandwidth is unlikely to impact the efficiency of persistent RNN kernels,

although other components of deep networks are known to be memory bound and would probably benefit.

5.2. Speech Recognition Task

This section explores the design space around very deep stacks of forward-only RNN models using a large vocabulary end-to-end English speech recognition task.

Figure 7 shows the architecture of the DS2 which we use as the baseline for these experiments: a recurrent neural network (RNN) trained to ingest speech spectrograms and generate text transcriptions using the CTC loss function (Graves et al., 2006). We evaluate various architectures by varying the number of recurrent layers and the number and span of skip connections between them. We use a dataset of 500 hours of audio in these experiments to quickly perform model design space exploration. We report Word Error Rate (WER) on an English speaker held out development set which is an internal dataset containing 2048 utterances of primarily read speech. We integrate a language model in a beam search decoding step as described in (Amodei et al., 2015).

Although the network architectures explored here are almost identical to the DS2 network, there is one significant difference. We use unidirectional RNN layers and row con-

Architecture	Dev (WER)
48 RNN, 61 total, no skip	100.0
48 RNN, 61 total, skip 1	38.77
48 RNN, 61 total, skip 2	33.28
48 RNN, 61 total, skip 3	30.32
48 RNN, 61 total, skip 4	29.40
48 RNN, 61 total, skip 5	29.82
48 RNN, 61 total, skip 6	30.04
48 RNN, 61 total, skip 7	29.87
48 RNN, 61 total, skip 8	27.44

Table 1: WER for models with skip connections added between every N RNN layers, each with 1152 units.

volutions (Amodei et al., 2015) with a fixed context size of 800ms rather than bidirectional RNN layers to make sure that the networks can be readily deployed in online speech recognition tasks.

5.3. Methodology

All models are trained for 20 epochs on the English dataset. We use stochastic gradient descent with Nesterov momentum (Sutskever et al., 2013) along with a minibatch from the range of $[64, 512]$ utterances. If the norm of the gradient exceeds the threshold of 400, it is rescaled to 400 (Pascanu et al., 2012). The model that performs the best on a held-out development set during training is chosen for evaluation. The learning rate is chosen from the range $[1 \times 10^{-5}, 6 \times 10^{-4}]$ to yield the fastest convergence and annealed by a constant factor of 1.2 after each epoch. We use a momentum of 0.99 for all models.

5.3.1. SENSITIVITY TO RESIDUAL CONNECTIONS

Table 1 shows the impact of residual skip connections on very deep RNN architectures with over fifty layers. We find that skip connections are essential for training these models, even when batch normalization is enabled. Models without skip connections fail to converge. For these networks, we find that skipping three or four RNN layers is substantially better than skipping a single layer, and moderately better than any other configuration (except for the outlier of 8, which we cannot explain). This suggests that residual skip connections enable effective optimization of very deep stacks of RNN layers.

5.3.2. SENSITIVITY TO DEPTH

Table 2 shows the impact of depth on residual RNN models with a constant number of parameters. We find that depth helps up to a point, about 50 layers, after which performance degrades. We hypothesize that this may be due to the thinning of individual layers in very deep networks.
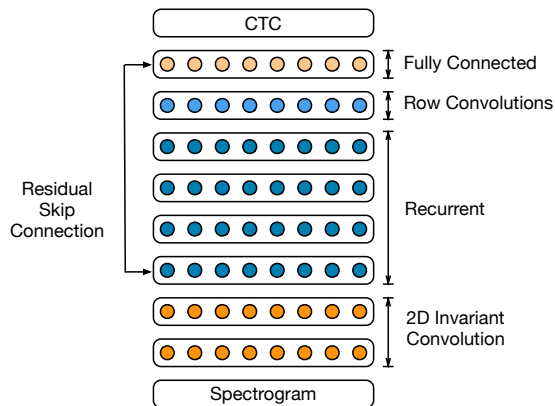

Figure 7: Architecture of the speech recognition system used in this paper. All networks use 2 layers of 2D invariant convolutions, and one fully connected layer. The basic module is a stack of four simple recurrent layers followed by a single row convolution layer, all of which may be bypassed by a single residual skip connection. Deeper networks are constructed by adding multiple of these basic modules. All networks use 10 layers of row convolutions. All layers except row convolutions use batch normalization.

5.3.3. SENSITIVITY TO PARAMETERS

Table 3 shows how training is affected by increasing the number of parameters in the network. We find that increasing the number of parameters by stacking additional layers generally improves performance of the network with continued gains out to the largest networks with approximately 100 layers and 200 million parameters.

5.3.4. SENSITIVITY TO MINI-BATCH SIZE

Table 8 shows how training is affected by increasing the mini-batch size. For these experiments, we perform a search over the learning rate, momentum, and annealing rate for each batch size to find a good value. We find that there is very little difference in the amount of computational work (in this case, the number of epochs) needed to converge for batch sizes below a threshold of 512 to 1024, but convergence is much slower beyond it. This suggests that even if GPU memory capacities were increased, running efficiently with a smaller batch size per GPU would enable data-parallel approaches to scale to more GPUs.

5.4. Discussion

In general, these results reinforce the trend of deeper models being more difficult to train, but delivering better performance if they can be trained successfully. We find both batch normalization and residual skip connections to be ef-

Arch	# Acts	Dev (WER)
8 RNN	1632	33.61
24 RNN	928	31.99
40 RNN	704	31.64
56 RNN	608	30.67
72 RNN	544	33.35
88 RNN	480	42.67

Table 2: WER on a training and development set for various depths of RNN. The number of parameters is kept approximately constant as the depth increases, thus the number of activations per layer decreases. For the architecture "M RNN" implies M uni-directional RNN layers.

Architecture	# Params	Dev (WER)
8 RNN, 21 total	22M	35.69
24 RNN, 37 total	65M	31.32
40 RNN, 53 total	107M	28.90
56 RNN, 69 total	149M	28.12
72 RNN, 85 total	192M	27.84
88 RNN, 101 total	234M	27.23

Table 3: WER on a training and development set for various depths of RNN. The number of parameters per layer is kept constant as the depth increases in this experiment, thus the number of parameters increases as the depth increases. For the architecture "M RNN, N total" implies M consecutive uni-directional RNN layers with N total layers in the network.

fective techniques that allow training deeper RNN models for speech recognition, reinforcing the importance of these techniques that has been previously demonstrated for CNNs applied to vision applications.

From a computational perspective, it seems clear that the development of faster processors with more memory capacity will likely enable even larger models to be trained on bigger data sets, unlocking additional accuracy. This work has shown that some model architectures are constrained not only by hardware performance, but also by the strategy used to map them to hardware. Mapping RNNs to GPUs using matrix multiplication is efficient for shallow networks with large layers, but persistent RNN kernels are much more efficient for very deep networks with relatively narrow layers.

In addition to work on model architecture exploration, and work on improving the performance of general purpose processors, it may also be fruitful to consider strategies for mapping currently inefficient model architectures onto existing hardware platforms. We have found the MBSP abstract machine model together with the guidelines in Sec-
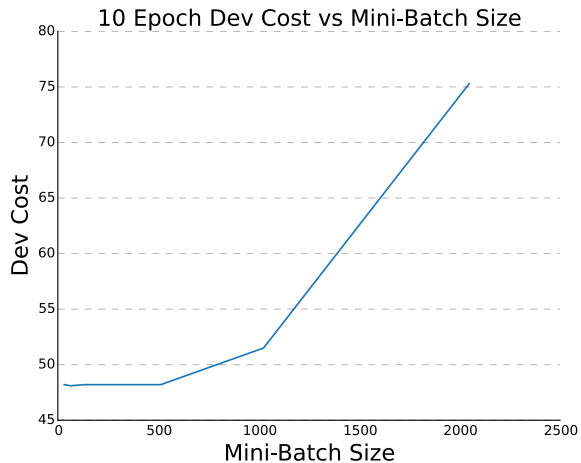

Figure 8: Development set cost after 10 epochs for various mini-batch sizes. The architecture is a 61-layer model with 2 layers of 2D-invariant convolution, 48 RNN layers (with 1152 activations and skip connections every 4 layers), 10 row convolution layers, and one fully connected feed forward layer. Note that the number of epochs needed to reach a given cost is approximately constant until the mini-batch becomes larger than 512-1024, at which point it grows considerably.

tion 3.1 to be a useful tool for quickly deciding whether or not a hypothetical model architecture can be efficiently mapped to hardware.

6. Conclusion

We demonstrate a technique for achieving high performance for RNN evaluation at very low batch sizes on an NVIDIA TitanX GPU, achieving 2.8 TFLOP/s at a mini-batch size of 4. This provides a 16x reduction in activation memory footprint, and allows us to train models with over 100 layers on the same hardware which is about an order of magnitude deeper than without this technique. We focus our evaluation on unidirectional RNNs with at most 800ms of future context, and demonstrate that accuracy continues to scale with increased depth. We expect these gains to directly enable the training of deeper RNN networks on much larger datasets than would be possible without this technique.

References

Abdel-Hamid, Ossama, Mohamed, Abdel-rahman, Jang, Hui, and Penn, Gerald. Applying convolutional neural networks concepts to hybrid nn-hmm model for speech recognition. In *ICASSP*, 2012.

Amodei, Dario, Anubhai, Rishita, Battenberg, Eric, Case,

Carl, Casper, Jared, Catanzaro, Bryan, Chen, Jingdong, Chrzanowski, Mike, Coates, Adam, Diamos, Greg, et al. Deep speech 2: End-to-end speech recognition in english and mandarin. *arXiv preprint arXiv:1512.02595*, 2015.

Bahdanau, Dzmitry, Cho, Kyunghyun, and Bengio, Yoshua. Neural machine translation by jointly learning to align and translate. *CoRR*, abs/1409.0473, 2014. URL http://arxiv.org/abs/1409.0473.

Bengio, Yoshua, Ducharme, RÃľjean, Vincent, Pascal, and Jauvin, Christian. A neural probabilistic language model. *JOURNAL OF MACHINE LEARNING RESEARCH*, 3:1137–1155, 2003.

Chen, Danqi and Manning, Christopher D. A fast and accurate dependency parser using neural networks. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*, pp. 740–750, 2014. URL http://aclweb.org/anthology/D/D14/D14-1082.pdf.

Chetlur, Sharan, Woolley, Cliff, Vandermersch, Philippe, Cohen, Jonathan, Tran, John, Catanzaro, Bryan, and Shelhamer, Evan. cuDNN: Efficient primitives for deep learning. URL http://arxiv.org/abs/1410.0759.

Collobert, R. and Weston, J. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *International Conference on Machine Learning, ICML*, 2008.

Dahl, G.E., Yu, D., and Deng, L. Large vocabulary continuous speech recognition with context-dependent DBN-HMMs. In *Proc. ICASSP*, 2011a.

Dahl, G.E., Yu, D., Deng, L., and Acero, A. Context-dependent pre-trained deep neural networks for large vocabulary speech recognition. *IEEE Transactions on Audio, Speech, and Language Processing*, 2011b.

Devlin, Jacob, Zbib, Rabih, Huang, Zhongqiang, Lamar, Thomas, Schwartz, Richard, and Makhoul, John. Fast and robust neural network joint models for statistical machine translation. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics*, volume 1, pp. 1370–1380, 2014.

Diamos, Gregory, Sengupta, Shubho, Catanzaro, Bryan, Chrzanowski, Mike, Coates, Adam, Elsen, Erich, Engel, Jesse, Hannun, Awni, and Satheesh, Sanjeev. Persistent RNNs. https://github.com/baidu-research/persistent-rnn. Accessed: 2016-05-23.

Dongarra, Jack, Gates, Mark, Haidar, Azzam, Kurzak, Jakub, Luszczek, Piotr, Tomov, Stanimire, and Yamazaki, Ichitaro. Accelerating numerical dense linear algebra calculations with gpus. *Numerical Computations with GPUs*, pp. 1–26, 2014.

Gao, Haoyuan, Mao, Junhua, Zhou, Jie, Huang, Zhiheng, Wang, Lei, and Xu, Wei. Are you talking to a machine? dataset and methods for multilingual image question answering. *CoRR*, abs/1505.05612, 2015. URL http://arxiv.org/abs/1505.05612.

Graves, A., Fernández, S., Gomez, F., and Schmidhuber, J. Connectionist temporal classification: Labelling unsegmented sequence data with recurrent neural networks. In *ICML*, pp. 369–376. ACM, 2006.

Graves, Alex, Mohamed, Abdel-rahman, and Hinton, Geoffrey. Speech recognition with deep recurrent neural networks. In *ICASSP*, 2013.

Gray, Scott. Assembler for nvidia maxwell architecture, 2014. URL https://github.com/NervanaSystems/maxas.

H. Sak, Hasim, Senior, Andrew, and Beaufays, Francoise. Long short-term memory recurrent neural network architectures for large scale acoustic modeling. In *Interspeech*, 2014.

Hannun, Awni, Case, Carl, Casper, Jared, Catanzaro, Bryan, Diamos, Greg, Elsen, Erich, Prenger, Ryan, Satheesh, Sanjeev, Sengupta, Shubho, Coates, Adam, and Ng, Andrew Y. Deep speech: Scaling up end-to-end speech recognition. 1412.5567, 2014. http://arxiv.org/abs/1412.5567.

He, K., Zhang, X., Ren, S., and Sun, J. Deep Residual Learning for Image Recognition. *ArXiv e-prints*, December 2015.

Hinton, G.E., Deng, L., Yu, D., Dahl, G.E., Mohamed, A., Jaitly, N., Senior, A., Vanhoucke, V., Nguyen, P., Sainath, T., and Kingsbury, B. Deep neural networks for acoustic modeling in speech recognition. *IEEE Signal Processing Magazine*, 29(November):82–97, 2012.

Iyyer, Mohit, Manjunatha, Varun, Boyd-Graber, Jordan, and III, Hal Daume. Deep unordered composition rivals syntactic methods for text classification. In *Association for Computational Linguistics*, 2015. URL docs/2015_acl_dan.pdf.

Krizhevsky, Alex, Sutskever, Ilya, and Hinton, Geoff. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25*, pp. 1106–1114, 2012.

Lavin, Andrew and Gray, Scott. Fast algorithms for convolutional neural networks. *CoRR*, abs/1509.09308, 2015. URL http://arxiv.org/abs/1509.09308.

Le, Phong and Zuidema, Willem. Compositional distributional semantics with long short term memory. *arXiv preprint arXiv:1503.02510*, 2015.

Mikolov, Tomas, Karafiát, Martin, Burget, Lukas, Cernockỳ, Jan, and Khudanpur, Sanjeev. Recurrent neural network based language model. In *INTERSPEECH 2010, 11th Annual Conference of the International Speech Communication Association, Makuhari, Chiba, Japan, September 26-30, 2010*, pp. 1045–1048, 2010.

Mohamed, A., Dahl, G.E., and Hinton, G.E. Acoustic modeling using deep belief networks. *IEEE Transactions on Audio, Speech, and Language Processing*, (99), 2011. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5704567.

N. Jaitly, P. Nguyen, A. Senior and Vanhoucke, V. Application of pretrained deep neural networks to large vocabulary speech recognition. In *Interspeech*, 2012.

Pascanu, Razvan, Mikolov, Tomas, and Bengio, Yoshua. On the difficulty of training recurrent neural networks. abs/1211.5063, 2012. http://arxiv.org/abs/1211.5063.

Sainath, Tara, Vinyals, Oriol, Senior, Andrew, and Sak, Hasim. Convolutional, long short-term memory, fully connected deep neural networks. In *ICASSP*, 2015.

Sainath, Tara N., rahman Mohamed, Abdel, Kingsbury, Brian, and Ramabhadran, Bhuvana. Deep convolutional neural networks for LVCSR. In *ICASSP*, 2013.

Sak, Hasim, Vinyals, Oriol, Heigold, Georg, Senior, Andrew, McDermott, Erik, Monga, Rajat, and Mao, Mark. Sequence discriminative distributed training of long shortterm memory recurrent neural networks. In *Interspeech*, 2014.

Seide, Frank, Li, Gang, and Yu, Dong. Conversational speech transcription using context-dependent deep neural networks. In *Interspeech*, pp. 437–440, 2011.

Simonyan, Karen and Zisserman, Andrew. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014. URL http://arxiv.org/abs/1409.1556.

Socher, Richard, Lin, Cliff C., Ng, Andrew Y., and Manning, Christopher D. Parsing Natural Scenes and Natural Language with Recursive Neural Networks. In *Proceedings of the 26th International Conference on Machine Learning (ICML)*, 2011.

Socher, Richard, Perelygin, Alex, Wu, Jean Y, Chuang, Jason, Manning, Christopher D, Ng, Andrew Y, and Potts, Christopher Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *EMNLP*, 2013.

Srivastava, Rupesh Kumar, Greff, Klaus, and Schmidhuber, Jürgen. Highway networks. *CoRR*, abs/1505.00387, 2015. URL http://arxiv.org/abs/1505.00387.

Sutskever, I., Martens, J., Dahl, G., and Hinton, G. On the importance of momentum and initialization in deep learning. In *30th International Conference on Machine Learning*, 2013.

Sutskever, Ilya. Training recurrent neural networks, 2013.

Sutskever, Ilya, Vinyals, Oriol, and Le, Quoc V. Sequence to sequence learning with neural networks. In *Proc. NIPS*, Montreal, CA, 2014. URL http://arxiv.org/abs/1409.3215.

Szegedy, Christian, Liu, Wei, Jia, Yangqing, Sermanet, Pierre, Reed, Scott, Anguelov, Dragomir, Erhan, Dumitru, Vanhoucke, Vincent, and Rabinovich, Andrew. Going deeper with convolutions. *CoRR*, abs/1409.4842, 2014. URL http://arxiv.org/abs/1409.4842.

Toshev, Alexander and Szegedy, Christian. Deeppose: Human pose estimation via deep neural networks. *CoRR*, abs/1312.4659, 2013. URL http://arxiv.org/abs/1312.4659.

Valiant, Leslie G. A bridging model for multi-core computing. In *Proceedings of the 16th Annual European Symposium on Algorithms*, ESA '08, pp. 13–28, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-87743-1. doi: 10.1007/978-3-540-87744-8_2. URL http://dx.doi.org/10.1007/978-3-540-87744-8_2.

Vasilache, Nicolas, Johnson, Jeff, Mathieu, Michaël, Chintala, Soumith, Piantino, Serkan, and LeCun, Yann. Fast convolutional nets with fbfft: A GPU performance evaluation. *CoRR*, abs/1412.7580, 2014. URL http://arxiv.org/abs/1412.7580.