Final Project: GPU Performance on Small Recurrent Neural Networks

David Brandfonbrener
CPSC 425

May 10, 2017

1 Introduction

1.1 GPUs and Neural Networks

One of the most important applications of GPU computing today is for neural networks [7]. A neural network works essentially by combining large linear transformations (ie matrix multiplication) with easily differentiated non-linearities. This creates a function that can represent nearly any nonlinear manifold that the programmer wants the network to learn to represent. Clearly, GPUs can help to speed up this type of computation by adding parallelism to the matrix multiplications.

As a result many python libraries have been developed and rigorously tested to get the best performance on very large tasks. Here we use Theano [10], a library in python that calculates the gradients that are so important to using the gradient descent algorithm to train a neural network. Gradient descent essentially finds a local minima in the space of possible parameters to the network that gives the lowest error on some predefined metric (here we use the mean squared error between network output and desired output). While there is much work on the effectiveness of GPUs on large neural network problems, there is much less literature about small neural networks.

Intuitively, the GPU should be less useful on smaller problems since there is less to be gained from faster matrix multiplication. This is especially true with recurrent networks, which feed output from one timestep as input into the next timestep, thus preventing parallelism across the timesteps since the calculations must be made sequentially. Additionally, how the data is loaded onto GPU will become an important factor. If the dataset can all be loaded into memory on the GPU, this would theoretically be much faster than loading the data piece by piece as it is used since the memory transfer is often slower than the operations being conducted.

Here we implement small neural networks with a consistent network architecture, but varying parameters. These parameters include different sizes of the networks, ranging from tiny networks that do not do more than a 10 by 10 matrix multiplication to larger, but still very simple networks. We also vary the number of simulated timesteps the network uses, as well as input and output size and type of nonlinearity. These factors are then also tested against a host of software and hardware variations to see how they interact. We test on multiple GPUs and CPUs and use both theano and tensorflow to conduct differentiation.

The structure of the paper is as follows. Section 2 describes the network and experimental setups in more detail. Section 3 presents data tables of the results of the experiments. And Section 4 provides a discussion of the various data.

1.2 Specific Connection to Neuroscience

Before I begin, I would like to add a brief note about my connection to this problem. I am working in a computational neuroscience lab that is working to train small and biologically inspired recurrent networks to accomplish small tasks. The reason to use recurrent networks is that the recurrent structure mirrors how real brains must operate in time, passing information from neuron to neuron. So, we have been training small networks, much along the lines of those found in [9] to perform simple tasks. We have mostly been using tensorflow [1] on CPU as we are developing the networks and are weighing the benefits of using the GPU more often. Unfortunately, tensorflow only runs on GPUs with Cuda capability 3.0 or above, so I used theano for most of this project so as to be able to run on the M2070 and M2090. But, in the end, I also implemented some tensorflow code on Grace. Some of the work for the neuroscience project can be found at https://github.com/davidbrandfonbrener/Project-Sisyphus, if you are interested.

2 Experimental Setup

2.1 The Network

The network I ended up implementing across various platforms is the simplest possible sequence to sequence recurrent neural network. The network takes as its input a sequence of length `n_steps` where the length of the sequence represents the number of timesteps in that sequence. At each timestep the input sequence contains a vector of length `n_in`. This vector is fed through the linear transformation defined by `W_in`, which is a `n_in, n_rec` matrix. Now, this vector of length `n_rec` is combined with the "hidden state" of the network from the previous timestep, and passed through the recurrent connection matrix `W_rec` which is a `n_rec, n_rec` matrix. Essentially, this means that we use our two matrices to combine inputs from the current timestep, with the "state" from the previous timestep. The "state" is where the recurrent connectivity of the network is able to store information about data the network has seen on past timesteps. So, for a perceptual task, this acts as a sort of short term memory for the network. We then apply the activation function, which in our networks for this project is wither a ReLU or a sigmoid function. The ReLU is defined as the element-wise function

$$relu(x) = \begin{cases} x & x > 0 \\ 0 & otherwise \end{cases}$$

while the sigmoid is defined as

$$\sigma(x) = \frac{1}{1+e^{-x}}.$$

The idea with the nonlinearity is to make it so that the network is not just making a big linear map. It is basic linear algebra that composing two linear maps gives you a linear map (ie. multiplying matrices yields another matrix). The nonlinearity adds complexity to the functions that the network can represent. After the nonlinearity, we have our new state vector. Now, to get the output at this timestep, which will be a vector of length `n_out` we just apply the linear transformation defined by `W_out` which is a `n_rec, n_out` matrix. To give out network more capability, we also change the transformations from linear to affine by adding biases `b_rec, b_out`, which are just vectors added to the state and output. This information is all encoded in the following equations:

$$\begin{aligned} state(t) &= relu(input(t)^T * W_in + state(t-1)^T * W_{rec} + b_{rec}) \\ output(t) &= state(t)^T * W_{out} + b_{out} \end{aligned}$$

To get a better understanding the following picture displays what a recurrent network looks like unrolled into timesteps. So, in the below figure, time progresses to the right and the red arrow corresponds to `W_in`, the green arrow to `W_rec`, and the blue arrow to `W_out`.

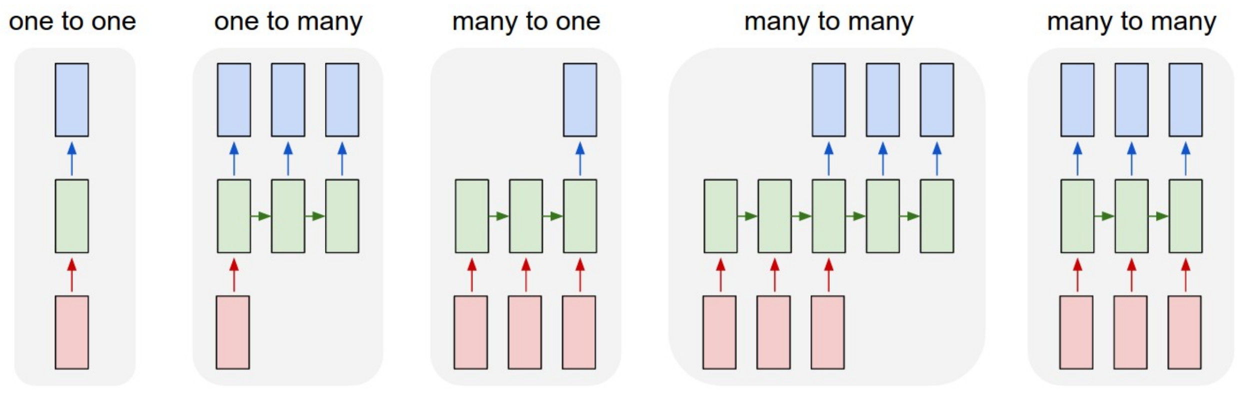

Figure 1: Various different ways design recurrent neural networks. Here we implement the last of these options [4].

The network trains to produce nearly the correct by calculating a loss function that is a function of the difference between the outputs of the network and the desired outputs. For these experiments, we always use the mean squared error, given by the sum of squares of the element-wise difference between network predictions and desired outputs. Then, that loss is viewed as an output of a function dependent on the values in the weight matrices, and we can calculate the gradients of the loss function with respect to the weights to decide how to change the weights to best minimize the loss function. Since we only calculate the gradients on small and randomly chosen samples of our data at a time, this is called stochastic gradient descent.

2.1.1 Data

For these experiments, we just used random datas. That is, the input and output data was generated element-wise uniformly at random from the interval (-5, 5). This is perhaps not the most realistic scenario since as there is no signal, the network may converge relatively quickly to just returning the mean of the distribution. But, the goal is just to understand how network design, hardware, and software effect the runtime of training. One problem that did result from the random data was exploding gradients, where the noisy data leads to rapidly fluctuating gradients that yield numerical precision problems. To combat this, I implemented gradient clipping as in [8] which seemed to do the trick and eliminate numerical problems. Additionally, it is worth noting that I used 32-bit floating point numbers throughout as GPU computations usually recommend this level of precision for better efficiency.

2.2 Hardware and Software

For these experiments, I used various hardware to get a comparison between the different machines. Most of the code was executed on an NVIDIA Tesla K80 GPU on the Grace cluster. This GPU has CUDA compute capability 3.7, which meant that it was able to run tensorflow (unlike the Omega cluster) since you need at least compute capability 3.0 to do so. The GPU has 2496 processor cores and a 560 MHz core clock speed and 24 GB of memory on board as well as a 2.5 GHz memory clock [5]. The Grace node with the GPU had an Intel Ivy Bridge CPU where each core has a 2.2 GHz clock speed and the machine has 10 cores [3]. Then, I conducted some trials on the Omega cluster, using an NVIDIA M2090 GPU. This GPU has 512 processor cores and a 1.3 GHz clock, as well as 6 GB of memory and a 1.85 GHz memory clock [6]. It is worth noting that when I conducted the experiments Omega was running quite slowly, it does not seem like this effected the results, but it is possible that it did. Lastly, I ran many trials on my Macbook, which is a 2016 Macbook Pro with a 2.9 GHz Intel Core i7 processor and 16 GB of memory.

Setting up the proper environments on the clusters was one of the most time consuming parts of the project. I installed anaconda locally on each cluster and then installed the various software libraries I needed from there. I used the modules on the clusters to use CUDA 8.0, CuDNN 7.5 (which only worked on Grace), and GCC 5.2. All the jobs on Grace were run through slurm batch scripts and the Omega jobs through torque scripts, all of this code is separated into the ”grace” and ”omega” directories in this project submission.

In terms of software, I coded all of the networks in python. However, the libraries that I used (both theano and tensorflow), are written in C and use python as an API. However, the way that python is compiled may have had some effect on results, as will be discussed below. Both theano and tensorflow are libraries that implement the calculation of gradients by constructing computational graphs that the compiler then determines how to differentiate [10] [1]. In addition to these libraries, I used a wrapper library called keras, which allows for easier construction of neural networks [2]. For example, what took me over 100 lines of code in theano, is less than 10 lines in keras. The keras library also claims to implement various optimizations in the code to increase performance from the naive implementation (essentially what I implement in theano). Moreover, keras runs on either a theano or tensorflow backend just by changing the `keras.json` file, so this allowed for easy comparison between the libraries.

2.3 Variables

The experimental question is to figure out which impacts are most important to the efficiency of these networks. There were three general types of changes that I examined. The most important to my research in the neuroscience lab were the network architecture questions, but the software and hardware questions were also interesting for this project in assessing the use of parallelism.

2.3.1 Network Structure

I varied the network size along a few dimensions. I kept the dataset a constant size of 1000 training examples, and trained on each example twice with a learning rate of .001. Other than that, I varied all of the other network parameters. This meant varying `n_in, n_rec, n_out, n_timesteps,` and the activation function. The most interesting variations to me were in `n_rec, n_timesteps` because these are in practice the most variable, and they also are the ones that cause the largest changes in runtime.

Another implementation detail that I came across along the way was how theano variables are defined. Variables can either be defined as shared or just as the numpy arrays. As shared data, if the GPU is in use, theano will preallocate and transfer the data to the GPU. So, I made one implementation that will do this and one that does not. In the case that the data is not all loaded onto GPU in advance, the data is ostensibly loaded one at a time, although since my datasets were normally very small, this effect was largely missed in these experiments because the data was likely loaded all at once anyways just because of various memory buffering sizes. Moreover, with the shared implementation, part of the function compilation is actually moved into the runtime. This will be discussed later.

2.3.2 Python Library

I wrote two different implementations in theano, as discussed above. Then, these implementations were compared with the keras optimized theano implmentation as well as the keras tensorflow implementation. Because the tensorflow would not run on all the platforms, the theano implementation was my main network used for experiments.

2.3.3 Hardware

I varied over GPU and CPU. I had access to the K80 and M2090 GPUs and the Ivy Bridge and i7 CPUs cited above. The interesting thing to test was how the hardware compared to each other and also whether there was any interaction between hardware and network structure or hardware and choice of python library.

3 Results

This section will present the timing results from the various experiments in a series of tables. Because I varied so many different variables in these experiments, there are numerous tables and visualizing the differences is often cumbersome. I have provided a few graphs of selected variation across different choices of just one variable, but this is by no means an exhaustive exploration of how the multitude of factors interacting with each other.

	`n_steps` = 10	20	50	100
`n_rec` = 10	20.9	13.0	22.9	39.3
20	9.7	13.1	23.0	39.6
50	9.8	13.2	22.7	39.1
100	9.8	13.0	22.7	39.5
200	10.2	14.0	25.2	44.2

Figure 2: Shared dataset. These times are taken from running on the Tesla K80 GPU on the Grace cluster using the pure theano code with ReLU activation and `n_in = n_out = 2`. As with all of our trials, we use 1000 training examples and train for 2 epochs with a learning rate of .001 and using gradient clipping.

	`n_steps` = 10	20	50	100
`n_rec` = 10	4.8	7.4	16.8	32.6
20	4.3	7.4	16.9	32.4
50	4.3	7.5	16.9	32.7
100	4.3	7.5	16.9	32.5
200	4.8	8.5	19.6	37.7

Figure 3: Unshared dataset. These times are taken from running on the Tesla K80 GPU on the Grace cluster using the pure theano code with ReLU activation and `n_in = n_out = 2`.

	`n_steps` = 10	20	50	100
`n_rec` = 10	14.6	12.5	21.6	36.7
20	9.3	12.6	21.4	36.5
50	9.4	12.5	21.5	36.5
100	9.4	12.3	21.4	36.1

Figure 4: Shared with sigmoid activation. These times are taken from running on the Tesla K80 GPU on the Grace cluster using the pure theano code with sigmoid activation and `n_in = n_out = 2`.

	`n_in=n_out` = 2	10	20	50	100	Macbook CPU `n_in=n_out` = 10
`n_rec` = 10	17.4	13.0	12.8	12.8	12.8	7.0
100	12.3	12.9	12.9	12.8	12.8	7.3
500	13.8	14.0	13.8	14.6	14.6	11.3
1000	22.9	18.4	18.5	18.3	17.9	18.6
2000	29.5	30.1	30.1	30.5	31.2	134.8
5000	108.8	109.3	108.8	107.8	108.0	834.1

Figure 5: Larger Matrices. These times are taken from running on the Tesla K80 GPU on the Grace cluster using the pure theano code with ReLU activation and `n_steps = 20`.

	`n_steps` = 10	20	50	100
`n_rec` = 10	45.6	9.3	14.7	25.6
20	7.0	8.8	14.9	24.9
50	7.1	8.8	14.7	25.3
100	6.9	8.8	15.1	24.2
200	8.4	10.2	17.1	29.7

Figure 6: Keras with Theano. These times are taken from running on the Tesla K80 GPU on the Grace cluster using the keras theano code with ReLU activation and `n_in = n_out = 2`.

	`n_steps` = 10	20	50	100
`n_rec` = 10	15.0	9.8	18.9	35.8
20	7.2	10.0	19.1	35.8
50	7.4	10.0	19.3	36.0
100	7.6	10.1	19.4	36.0
200	7.8	10.7	20.8	39.0

Figure 7: Keras with Tensorflow. These times are taken from running on the Tesla K80 GPU on the Grace cluster using the keras tensorflow code with ReLU activation and `n_in = n_out = 2`.

	`n_steps` = 10	20	50	100	200
`n_rec` = 10	89.4	22.4	40.6	70.7	130.1
20	16.5	22.5	40.7	75.5	130.7
50	16.5	22.5	40.7	71.5	217.8
100	16.5	22.6	40.9	71.4	131.3
200	17.4	24.4	45.0	80.0	148.75

Figure 8: Omega M2090 GPU. These times are taken from running on the M2070 GPU on the Omega cluster using the pure theano code with ReLU activation and `n_in = n_out = 2`.

	`n_steps` = 10	20	50	100
`n_rec` = 10	14.1	8.2	12.5	19.7
20	6.9	8.4	12.9	20.0
50	7.0	8.5	13.2	21.7
100	18.6	29.2	49.7	113.2
200	23.1	37.2	82.2	154.2

Figure 9: Grace CPU. These times are taken from running on the CPU on the Grace cluster (this is an Intel Ivy Bridge CPU) using the pure theano code with ReLU activation and `n_in = n_out = 2`.

	`n_steps` = 10	20	50	100
`n_rec` = 10	6.8	6.6	7.8	8.6
20	7.6	9.9	10.1	12.3
50	8.5	8.1	8.9	9.5
100	6.8	7.1	8.4	10.2
200	7.1	7.5	9.1	12.2

Figure 10: Macbook CPU. These times are taken from running on my 2016 Macbook Pro using the pure theano code with ReLU activation and `n_in = n_out = 2`.

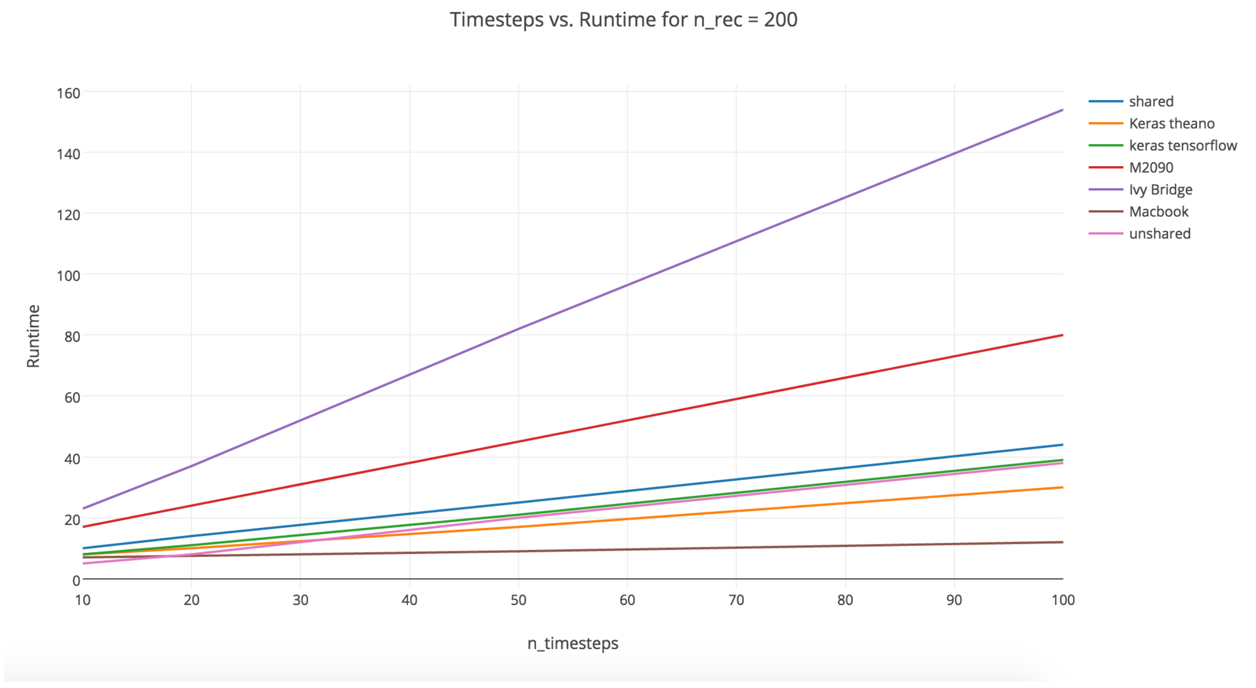

Figure 11: Plot of increasing timesteps for various software and hardware combinations.

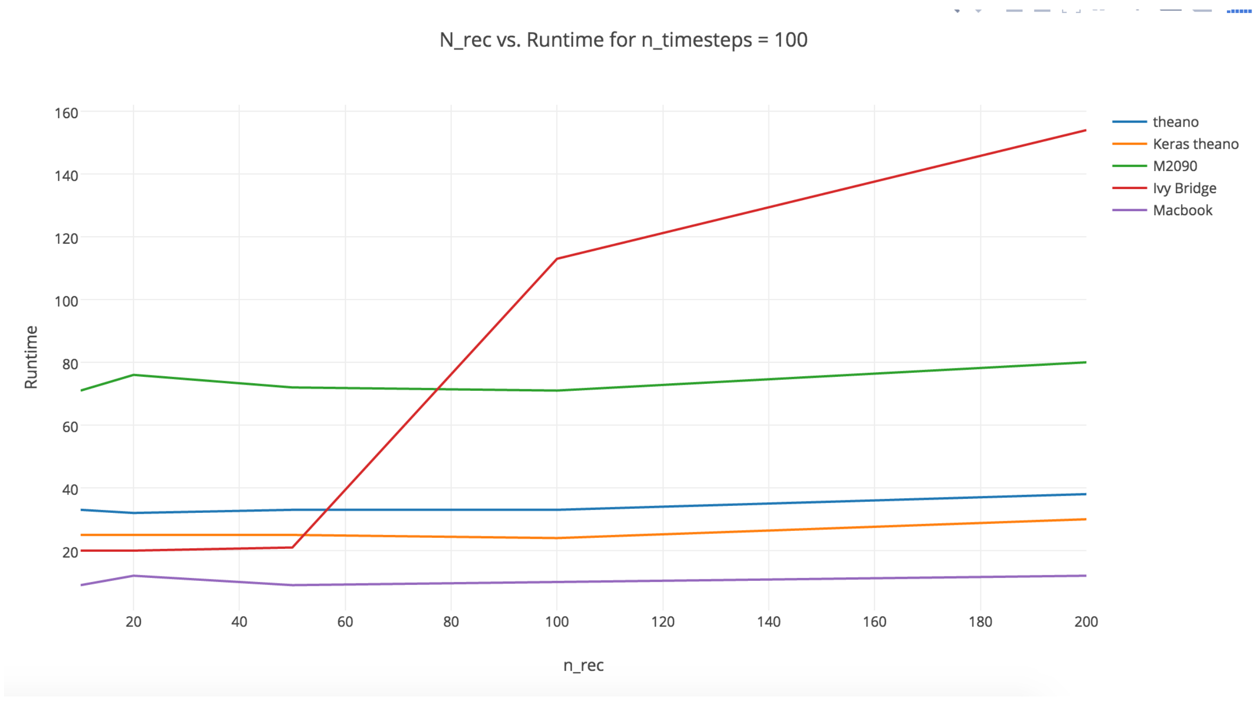

Figure 12: Plot of increasing recurrent size for various software and hardware combinations.

4 Discussion

This section will discuss the above results to explain the performance that we saw for various changes. In the end, we will see that different implementation choices will work better for different use cases, there is no truly dominant strategy.

4.1 Network Architecture

4.1.1 Shared vs. Unshared

One decision, that is very theano specific, is whether to store data in shared memory. With large datasets, looped over many times, and small enough to fit on GPU memory, shared memory could clearly be very beneficial as moving data onto the GPU can be a bottleneck. However, in my experiments, as seen in Figures 2 and 3, the unshared version of the code actually operated faster, by about 6 seconds across all trials. I have two possible explanations for this. First, assuming that there is no extraneous factor at play, it may be that loading the data into memory is actually inefficient at the beginning. If the loading is fast enough, some pipelining is achieved by loading data while other computations are happening, rather than waiting to load all the data at the beginning. Since these operations are somewhat computationally intensive, maybe the computation is the dominant cost. Another explanation could be that there is an extraneous impact of the way theano compiles functions. A theano "function" will be compiled when it is added to the computational graph. In my shared data implementation, this does not happen until the "train" function is called (because the "sgd_step" function is defined in the "train" function rather than the "RNN" constructor). This is also when timing starts. As a result, the timing may be factoring in some compile time. This could make sense because the timing gap is relatively consistent across different size inputs.

4.1.2 Timesteps

The most important and clear result of these experiments was that timesteps is by far the most important dimension on GPU. The way that the recurrent connection is defined, the timesteps cannot be parallelized because each one relies on the output of the previous one. In the theano code, this means that we need to use the "scan" function, which replaces "for" loops in the theano computational graph. We see linear scaling in the number of timesteps across all implementations, as seen in Figure 11. This makes sense because increasing the length of a loop without having parallelization should linearly increase the run time. The slopes of these lines are usually a little bit less than 1, as would be expected by just repeating one computation more often, while the rest of the program remains the same. So, large numbers of timesteps will render the GPU relatively ineffective since its main advantage is parallelization. The variation comes on the CPU, which I will address more in the below hardware section. Here we can notice that the Grace CPU performed generally poorly with `n_rec = 200`, but with smaller recurrent matrices, the Grace CPU and my Macbook CPU both outperformed the GPU. The CPU has faster clock times that allow it to execute the serial loops faster, but the GPU is able to parallelize the matrix multiplications more so performs better with larger matrices.

4.1.3 Neurons/Hidden/Recurrent Units

Varying `n_rec` is where the GPU was able to show its value. With large number of recurrent units, we see the GPU performing much better than the CPU. Even my Macbook, which had incredible performance on the smallest tasks because it has the fastest processor, was much slower than the GPU when the matrices became large. With a constant number of timesteps (read loop iterations), the GPU scales in the size of the recurrent matrix almost linearly, and better than the quadratic big O of matrix-vector multiplication (which should be the dominant operation). In Figure 5, I present the results for larger matrices. Now the GPU outperforms the CPU beginning at a 1000x1000 recurrent matrix. And, by the time we make the recurrent matrix 5000x5000, the GPU is giving us a nearly 8x speedup, which is very dramatic. This makes sense because multiplying large matrices can really benefit from parallelism, as we saw in assignment 6. So, while CPU had an easier time with the longer loops, one the recurrent matrix becomes large enough, the CPU cannot handle the computation that it must do in a more serial manner than the GPU.

4.1.4 Input and Output Sizes

The input and output sizes did not have much of an effect on performance. Again Figure 5 shows this experiment for different input sizes there is essentially no variation in performance. This makes sense because the Rectangular matrix multiplications from `W_in, W_out` are dominated in terms of runtime by the square matrix `W_rec`, since the recurrent matrix (in my applications) will be larger than the inputs and outputs.

4.1.5 Activation Function

One other interesting result was that the sigmoid activation function slightly outperformed the ReLU activation. This can be seen by comparing Figures 2 and 4. Here we see an improvement of 1-3 seconds depending on the overall runtime, suggesting that the sigmoidal units allow for faster calculations. This means that theano is calculating the gradients of these functions slightly faster than the ReLU ones. This is somewhat surprising since the ReLU is piecewise linear, so its derivatives are very easy to calculate. However, the difference must stem from testing the conditional and then returning the gradient being slower than having a non-piecewise function like the sigmoid. Also, since the sigmoid used to be more popular, there is likely some good optimization for calculating it. Again, this result was small, but consistent and relatively interesting.

4.2 Software Platform

The software platforms compared were my theano code with the keras theano code and the keras tensorflow code. Comparing the results from keras in figures 6 and 7 with my results in figure 3, we can see that my unshared data code outperformed the keras implementations when the number of timesteps was small. However, when the umber of timesteps reached 50, the keras theano implementation outperformed my theano code. The keras tensorflow implementation was consistently the worst. This makes sense for a few reasons. First, my theano code is much lighter than the keras version. Keras provides much more functionality, but as a result has many more conditionals and potential optimizations that it must check at every iteration. On the other hand, my code is as bare bones as possible to try to isolate what is causing any timing differences. However, for larger runtimes, keras is actually running faster. Keras implements more optimizations to deal with larger problems that allow their code to scale better than mine does. For example it only scans through the state vectors rather than the states and outputs, which may save time on larger scans, or preprocess the input with `W_in` all at once, rather than as part of the scan. By removing code from the slowest part of execution, the keras code scales better in the worst case of larger number of timesteps. Tensorflow's worse performance is not too surprising as it is known to be a little slower on GPU than theano at times. It could be interesting to explore how the implementations differ at the source code level, but I did not have time to investigate this for this project.

One other little thing to notice was that in many of the experiments the very first trial (in the upper left of the table) had strangely long runtime. This is likely to due to some sort of compile issue, or an issue of loading data or libraries or something of that nature. Importantly, I did not see this issue running coded on my laptop, but did see it on every cluster experiment except the unshared dataset, but that trial was run in close succession after another one. This did not end up really effecting my results much, but is something that I have had trouble explaining.

4.3 Hardware

4.3.1 GPU vs. CPU

The main goal of the project is to see the effect of parallelism of GPUs. The GPU is one of the main reasons for the growing importance of neural networks. However, here we have found a use case in which very small recurrent neural networks perform better on CPU than GPU. To see the GPU performance advantage, we needed to make the network noticeably larger, but then were able to get an 8x speedup, which is what GPUs promise. However, because the loop in a recurrent network has a dependence, it prevents parallelization, giving the speed of the processor more importance than how many processors there are. This tradeoff was made very clear when my laptop was able to outperform the GPU on certain tasks by up to about 3x. This should be an important reminder that while GPUs are powerful computational tools, they only help for certain types of problems. Namely, they are not good at executing loops that cannot be parallelized because they do not have fast processors, just many processors.

It is important to remember that GPUs are much better for larger networks, and for non-recurrent networks. For example, convolutional networks used for image processing are useful almost because GPUs allow them to be feasible by increasing parallelism. But, for

4.3.2 K80 vs. M2090

The K80 was dramatically better than the M2090. This makes sense as the K80 is much newer so that tensorflow will not even run on the M2090. The M2090 scales in a similar way to the K80, where the size of the matrix multiplication does not have much effect on runtime for relatively small values, but scaling in timesteps is nearly linear of slope 1 from the beginning. As I said above, Omega was running quite slowly when I performed these tests, which may have had some effect, but it was also to be expected that the M2090 ran much slower than the K80.

4.3.3 Macbook vs. Cluster CPU

The strangest results of the project were that my laptop ran the fastest out of any of the hardware options on the smaller inputs. There may have been other factors at play in terms of the way that libraries were loaded and whatnot, but the main explanation I have for this is that my laptop had the fastest single processor out of any of the hardware being tested. So, for the most serial of calculations, it could be expected that my laptop would have the best performance. The cluster CPU was especially slow in comparison for matrices of larger sizes. Essentially, in these experiments we saw CPU being much better for more timesteps and smaller matrices and then at a certain matrix size, about 1000 on my laptop and 100 on the cluster, the CPU began to run dramatically slower, allowing the GPU to surpass it in efficiency. This is especially clear in Figure 12, which nicely shows where the change kicks in. That said, this is one issue that it would be worth looking into further for alternative explanations.

4.4 Future Directions

This report gave me some helpful results about the tradeoffs between various network architectures and implementation decisions regarding python libraries and hardware. There are some other experiments that I now see could be interesting to pursue in the future to even better understand those tradeoffs.

For example, batching results in the gradient descent algorithm is often used to increase performance. I did not implement this strategy here, but it would be expected that the GPU could be more effective with larger batches since there is effortless parallelism created across a batch. Along the same lines, SGD is not the only way to optimize network weights. It is the simplest which is why i used it here, but it would be interesting to see if more complicated algorithms could lead to more variation across the various implementations.

Another change would be to use potentially more realistic training scenarios. For example, I did not use any real data, but just randomly generated data, so the network got noise in and was expected to produce different random noise. There is clearly no real learning going on for the network here and it is likely to converge to a simple solution of just returning the mean always. This could potentially effect computation time if the network learns a set of parameters with many zeros, and the hardware has optimizations to speed up calculations with zeros, so a real dataset may provide more intricate results. Along the same lines, in real applications the training is often run for much longer than I ran it in these experiments. It is possible that with more repetitions of the training algorithm, you may see larger effects from memory and instruction caching and the like which were missed in these shortened experiments.

Lastly, it would be interesting to extend similar tests to different kinds of networks (ie not just basic small RNNs) to see how they would extend. That said, small RNNs seem to at least theoretically be the worst case for GPUs since they allow for the least parallelization, which is the main result of this project.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng, *TensorFlow: Large-scale machine learning on heterogeneous systems*, 2015. Software available from tensorflow.org.

[2] Franois Chollet, *keras*, GitHub, 2015.

[3] Intel, *http://ark.intel.com/products/75272/intel-xeon-processor-e5-2660-v2-25m-cache-2_20-ghz.*

[4] Andrej Karpathy, *The unreasonable effectiveness of recurrent neural networks, http://karpathy.github.io/2015/05/21/rnn-effectiveness/.*

[5] NVIDIA, *https://images.nvidia.com/content/pdf/kepler/tesla-k80-boardspec-07317-001-v05.pdf.*

[6] ——, *http://www.nvidia.com/docs/io/43395/tesla-m2090-board-specification.pdf.*

[7] Kyoung-Su Oh and Keechul Jung, *Gpu implementation of neural networks*, Pattern Recognition **37** (2004), no. 6, 1311–1314.

[8] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio, *Understanding the exploding gradient problem*, CoRR **abs/1211.5063** (2012).

[9] H. Francis Song, Guangyu R. Yang, and Xiao-Jing Wang, *Training excitatory-inhibitory recurrent neural networks for cognitive tasks: A simple and flexible framework*, PLOS Computational Biology **12** (201602), 1–30.

[10] Theano Development Team, *Theano: A Python framework for fast computation of mathematical expressions*, arXiv e-prints **abs/1605.02688** (May 2016).