Autogenerating Text via Parallel Implementations of Recurrent Neural Networks

Final Report

Taylor Clarke (tclarke)
Anirban Ghosh (anirbang)

I. Introduction

a. Problem

Neural networks are a machine learning method to predict an output variable based on a set of input variables. Usually many points are used to "train" the neural network to attempt to compute the correct output, and then the network is tested on other data with known outputs to estimate its accuracy. If the neural network meets the specifications for the precise needs of the developer, it can then be used to predict outputs for data that is unknown. A neural network is essentially a directed graph, where a node computes a function of its input and directs that output either to another node or to be the final output of the network. The functions represented by the nodes can be univariate or multivariate and produce anything from a basic linear function to a binary classifier of its input. With the appropriate choice of functions, a neural network can be trained to produce a wide range of outputs.

Our specific interest is a recurrent neural network, which is merely a neural network where some of the output of the nodes are fed back to nodes previously visited, ie there is a cycle in the network. Recurrent neural networks are often used when we suspect that the output is correlated to other outputs in the dataset, so that utilizing a function of previous input and output will give information about the next output. One common way to visualize this is a time series, such as stock data. We expect the price of a stock at a certain time of day to depend not only on the time of day but also on the prices at previous timesteps. This also means that recurrent neural networks are an obvious tool for natural language processing, where previous characters in a word have some effect on the next character, previous words in a sentence have some effect on the next word, and so on. This is the foundation for our task. The goal of this process is that upon seeding the network or relying upon the network to seed itself, we allow the network to use its own output as the next input, so it could theoretically produce a large amount of meaningful text, code, or data on its own [1].

Parallelization of neural networks has been a hot topic in academia and industry for a number of years. As with any graph problem, parallelization is not obvious or easy, but can lead to great speedups when implemented cleverly. The major issue with parallelizing neural networks is that in order to train the network effectively, errors must be backpropagated through the network to compute the parameter alterations. This means that information must be passed both forward and backward through the net, leading to a large amount of interdependence and communication in the process. However, if independent nodes, or independent batches of

training data, can be computed in parallel and data communication costs can be minimized, a parallel algorithm has potential to be much faster than a sequential algorithm [2].

b. Approach

Our project consists of two major portions: (1) attempt to run an RNN on the Xeon Phi, and (2) study the differences between two existing RNN implementations and attempt to utilize our knowledge of parallelization to optimize the two different RNN implementations. Both of these implementations parallelize the RNN via data parallelism, ie they train the neural network by asynchonously processing the training data, with each asynchronous thread receiving a portion of the training data to process. This is in contrast to model parallelism, where the forward propagation to predict the next output and the back propagation of prediction error and gradient adjustment are processed in parallel, either by nodes being processed in parallel, layers being processed in parallel, or an even more complicated algorithm.

c. Related Work

Our work is based off of a few packages. First, the RNN we attempted to run on the Xeon Phi came from the Char-RNN package built by Andrej Karpathy [3]. Details on this package will come later, but it is important to note that this package utilized the scientific computing package Torch, which was the crux of the effort to run Char-RNN on the Xeon Phi. Second, we chose Faster-RNNLM to compare to Char-RNN [4]. Faster-RNNM was developed by the company Yandex. We initially looked at Faster-RNNLM because it was developed from scratch in C++, only utilizing the external package eigen to perform linear algebra operations. Eigen is also built in C++. With these packages built in C++, we were able to try different methods of getting them to run on the Phi, as compared to Char-RNN which required Torch, built in C and Lua. When we were ultimately unsuccessful at getting Faster-RNNLM to run on the Phi, we decided that since it was sufficiently different from Char-RNN that we could use them as exemplars to compare and optimize. Each of these packages comes from a large corpus of development and research into various RNNs and methods to optimize them.

d. Contributions

We made great strides in attempting the various methods available to us to run an RNN on the Xeon Phi and have produced a summary of the various methods and their pros and cons. We also analyzed the two different RNN packages, Char-RNN and Faster-RNNLM, and have suggested further optimizations these packages could utilize.

II. Methods/Approach

a. Xeon Phi

The Xeon Phi is “Intel’s first bootable host processor specifically designed for highly parallel workloads,” being an accelerator with 64 - 72 cores capable of running x86-64 code natively or “offloaded” from a host program [5]. A Xeon Phi is an interesting middle ground between a GPU and a CPU, running fewer threads than a GPU but being more portable due to

running C++ code as well as OpenCL code. We attempted to use a Xeon Phi because we wished to compare the performance of an RNN on the Phi to its performance on a GPU or CPU.

Our initial goal was to alter an existing RNN to run on the Xeon Phi. We chose Char-RNN for this goal since we obtained most of our information into text-based RNNs from a web post written by Andrej Karpathy, who developed Char-RNN. Char-RNN was developed to run on both a CPU and a GPU with CUDA, so we assumed that the parallel structure of the algorithm, even if not optimal, could be easily translated into a working method for the Xeon Phi.

Char-RNN utilizes the framework Torch, which is "a scientific computing framework with wide support for machine learning algorithms" [6]. It is often used specifically to implement neural networks in research and in industry. Char-RNN is a simple piece of Lua code written to implement existing neural network and CUDA packages already built in Torch, in order to learn character associations from a large text file and then autogenerate text based off of that training data. When taken together, the Char-RNN package along with the entire corpus of Torch code that it utilized proved to be a large codebase. Thus, instead of attempting to offload certain pieces of code, we looked into various ways of running the Torch framework on the Phi. Those various methods are enumerated below.

i. OpenCL

Our first attempt to run Char-RNN on the Xeon Phi was based around getting Torch to run in OpenCL instead of CUDA. OpenCL being an open source version of CUDA, less optimized but potentially more generic, we were able to find many sources saying that Torch could run on OpenCL. Thus, we were able to find a version of Torch cited in a few places that was built specifically to run OpenCL instead of CUDA [7].

On the other hand, the Xeon Phi was not able to run OpenCL automatically. There proved to be less information into this specific problem, since relatively few researchers use the Phi, and the subset of those who needed to run OpenCL on the Phi was even smaller. However, we were able to find instructions on running OpenCL on the Phi provided by Intel. To run OpenCL on the Phi required installing an SDK as well as a runtime environment provided by Intel. In attempting to install these tools, we found that the operating system versions supported by these tools were ahead of the version installed on our computing nodes. Therefore, we abandoned this approach, although we believe that with the correct operating system, this approach holds the most promise to running Torch on the Phi.

ii. Colfax Research

Colfax Research is a research company devoted to computing, and much of their research is in parallel optimizations of machine learning [8]. This company has greatly advertised their strides in running machine learning algorithms on the Xeon Phi and have produced publications and presentations into the matter. They have a Github repository that is forked from the main Torch branch with various commits, and they hint that they have altered Torch sufficiently to be able to run it on the Xeon Phi. Thus, this seemed like an obvious approach for our goal. However, after much deep diving through their code, commits, documentation, and blog posts, we found that their public version of Torch could not be

compiled for the Phi; it was merely built to test the neural network that they had trained on the Phi and that they have provided publicly.

iii. Intel

While skimming through various posts and documents, we found that Intel had developed a version of Torch that could be run on Intel computing architectures, and they specifically noted the Xeon Phi as a target architecture [9]. If we could mate this version of Torch with Char-RNN, then we theoretically would be able to run Char-RNN on the Phi. However, after many attempts at compilation and downloading various versions of different packages, we found that the version of Torch supplied by Intel was not under active development and had not been kept up to date with Torch, which is constantly under active development. Therefore it was not able to link with the neural network packages required by Char-RNN.

iv. Faster-RNNLM

After the many deep dives down various paths to run Torch on the Xeon Phi, we decided to take a step back and see if we could customize native C++ code ourselves to run on the Phi. We found the Faster-RNNLM package to be the most analogous to Char-RNN, the only outward difference being that it trained on words instead of characters. Otherwise, it could take in the exact same text file and produce output that was directly comparable to the output of Char-RNN.

We had two options to run on the Xeon Phi: natively and via offloading. Running natively requires that most of the time spent in the algorithm is in highly parallelizable tasks with little communication due to the small amount of memory and speed available to one thread on the Phi. Therefore we attempted going down the route of offloading parallel portions of code from the compute node running Faster-RNNLM. This method requires that all relevant data needing to be accessed by the parallelized threads be offloaded from the compute node with specific macros. We began the task of collecting the data and transforming it into a structure available to be offloaded. Much of this effort focused on transforming the input training data. Faster-RNNLM was developed such that each thread trains on a portion of the input text file, so that multiple threads access the text file throughout the entire training process of the neural network. Thus, this training data had to be put in a format available to be used on the Xeon Phi. After much effort in transforming the text file and the text parsing algorithms to be used on the Xeon Phi, we realized that the instance of the neural network class itself was a large class with many functions and instances of other classes within it. This was an issue because any data transferred to the Phi must be bitwise copyable, so we could not easily transfer this neural network object to the Phi [10]. Therefore we were forced to use the transformations of the text file as an experiment on the CPU rather than a method of letting the Phi run the algorithm.

b. Char-RNN vs. Faster-RNNLM

Although we had spent much time in attempting to get an RNN to run on the Phi and learned quite a bit about all the various hardware and software involved, we did not have many clear results to show. Therefore we took a step back and decided that we should focus our efforts on studying the existing implementations of Char-RNN on a GPU and CPU, and

Faster-RNNLM on a CPU. We studied their implementations, tested various optimizations, and profiled them in order to propose various improvements.

III. Experimental Setup

a. Software

i. Char-RNN

As mentioned before, Char-RNN is a package based on Torch, built by Andrej Karpathy. Therefore it was necessary to download the Char-RNN Github repository and follow the instructions in the readme provided in that repository in order to install Torch. No extra work was required to be able to run this code; merely following the instructions given in the readme was adequate to train the neural network and produce samples from it. For a baseline, we trained on a ~1MB text file of concatenated Shakespeare text with a neural network of 2 layers, 128 nodes per layer, 50 batches being processed in parallel, and 50 epochs (number of traversals through the training data). Then, when sampled with the defaults provided in the code and seeded with the letter "I", the neural network produces samples like the following:

I' have desinutenate are Which eye onure to think

ii. Faster-RNNLM

As previously stated, Faster-RNNLM is a package build in C++ by the technology company Yandex. It is meant to be a general framework for the public to train a wide variety of types of recurrent neural networks in order to generate words from a text file. It requires that each line in the training file consist of a single sentence and that there be no empty lines. Other than those stipulations, one can reproduce our data by downloading the Github repository there and simply following the instructions there. We trained on the Shakespeare data with all empty lines removed and with 1 neural network layer, 128 nodes, hierarchical softmax as the output layer, and a learning rate of 0.01. When sampled with the default settings given in the program and seeded with the letter "I", the neural network produces samples like the following:

I know so. DUCHESS they think shame, And man, the shout didst the queen to the curtsy
I the sink hast state never will greater the sweet well-disposed
I had kneel to on't. have never to their alone with me, sir?
I live:
I saw God LUCENTIO: me awake and my other

b. Hardware

We trained on both CPUs and a GPU, utilizing the Latedays cluster for our results. Each node on Latedays consists of the following in terms of CPUs:

- Two, six-core Xeon e5-2620 v3 processors
- 2.4 GHz, 15MB L3 cache, hyper-threading, AVX2 instruction support

- 16 GB RAM

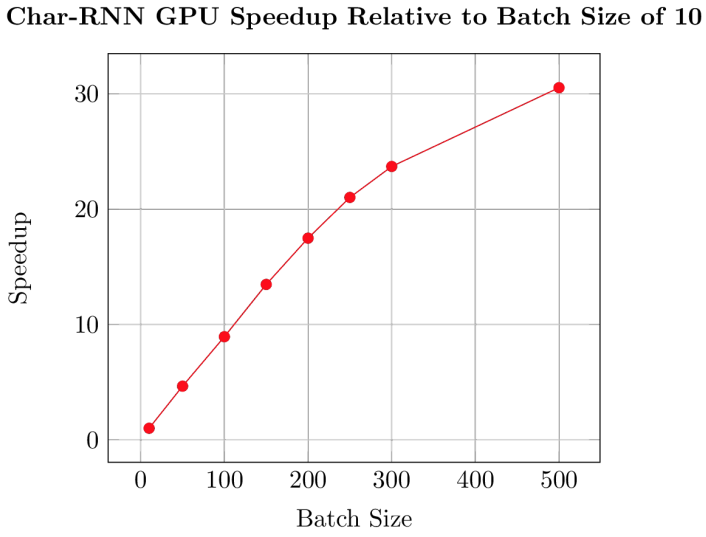
And the GPUs on Latedays consist of the following:
- A NVIDIA K40 GPU (4.3 TFLOPS)
- 12 GB RAM (288 GB/sec memory bandwidth)
- CUDA Capability 3.5
- 15 SMs

Since we attempted to run on the Xeon Phi, the specs for the Phi are:
- A 60-core Xeon Phi 5110P (2 TFLOPS)
- 60 cores (1 GHz, 4-threads per core, AVX512 ("16-wide") instruction support)
- 8 GB RAM (320GB/sec memory bandwidth)

IV. Experimental Evaluation

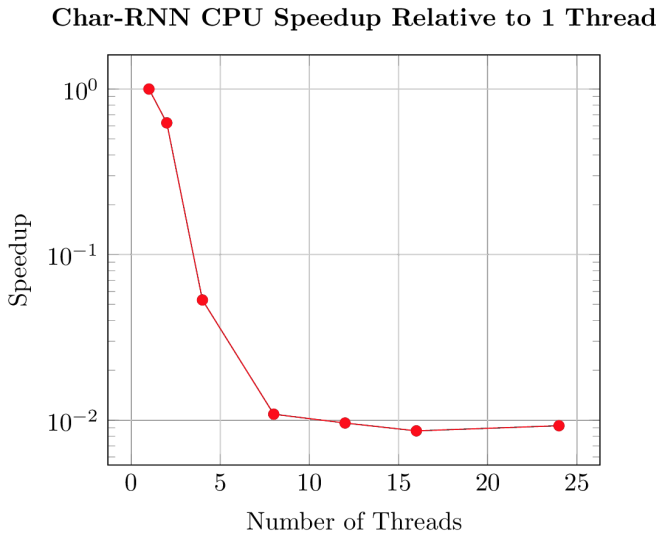
a. Char-RNN

The first experiments we produced involved testing Char-RNN on the GPU to examine the efficiency of its parallel structure. Since it was developed to utilize Torch, which is famously very efficient in CUDA, Char-RNN achieves a nearly perfect speedup as batch size is increased. Batch size refers to the number of text streams being processed in parallel. The speedup increases nearly linearly until about a batch size of 250, which is where interprocess communication between the batch processes in order to update the neural network begin to become a bottleneck. A graph of the speedup on a GPU is shown below.

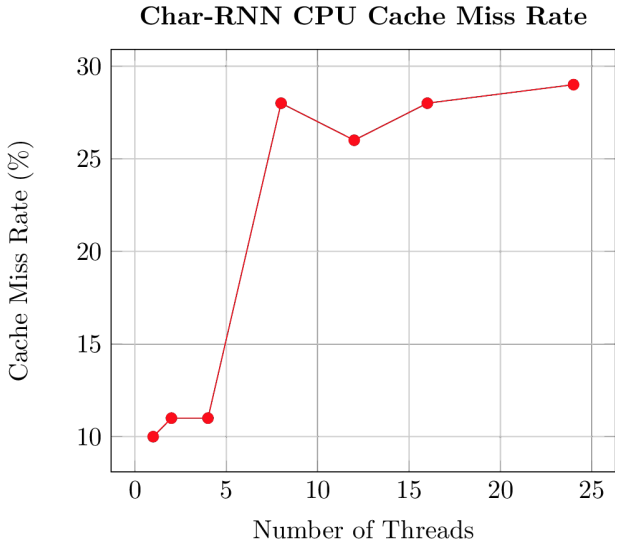


We find that this speedup is directly in opposition to its speedup when run on the CPU. Even though Char-RNN parallelizes its training over multiple batches of training data, it processes gradients and performs linear algebra utilizing CUDA threads at a much finer granularity. On the other hand, when the program is run specifically without utilizing a GPU,

Torch utilizes OpenMP to parallelize tasks. Due to the fact that we are already parallelizing over batches, this causes an extreme slowdown when Char-RNN is run on a CPU. A graph of this effect is shown below.



We can see that the slowdown here correlates to the cache miss rate, shown below. Due to Torch attempting to parallelize linear algebra and other tasks over OpenMP while the threads are already processing individual batches of training data in parallel, the cache miss rate increases drastically as the number of threads increases. It also is affected by the system spreading the computation across multiple physical cores, since the NUMA system must invalidate the caches across the cores.



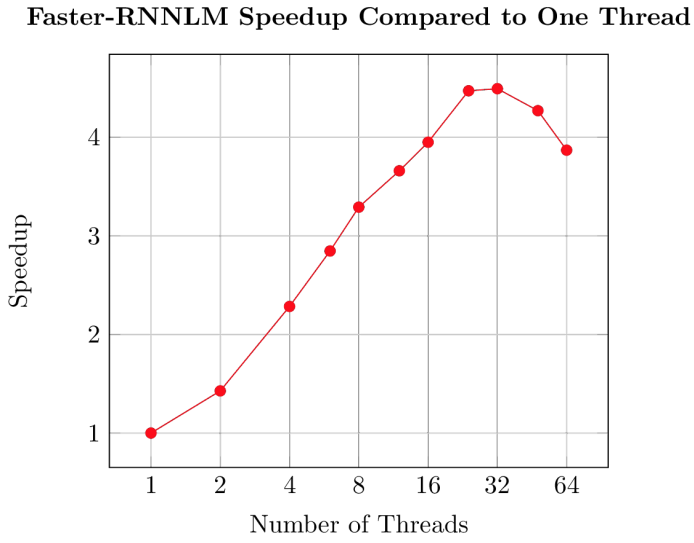
In addition to these findings, we conducted several experiments to test our conclusions. We extensively benchmarked and profiled the Char-RNN package and quickly realized that the

default implementation does not effectively exhaust the GPU resources. This was made clear by the fact that the same code ran as fast on a laptop's mobile GTX920M as it did on the Latedays cluster's Tesla K40M GPUs.

In order to speed-up the training time, we first tried fine grained parallelism in the BLAS libraries, for both the CPU and GPU implementations of Torch. However, this does not help much since the BLAS libraries are already highly optimized by the manufacturers and the Torch community. Secondly, upon profiling we found that most of the mathematical operations only consist of tight loops with with simple mathematical operations. These functions are called hundreds to thousands of times per iteration. Fine grained parallelism does not help because the sheer overhead of spawning and joining threads dominates over any benefits provided by parallelism.

b. Faster-RNNLM

Faster-RNNLM was built in C++ utilizing pthreads to parallelize over training data, with each pthread getting an equally sized portion of the training file to read. The most interesting aspect of Faster-RNNLM was that it used an idea called Hogwild, which states that no synchronization is necessary when training a neural network in parallel due to the low likelihood that two threads will be altering the same parameters at the same time [11]. This is the case because with random training data, no threads will be observing the same text features at the same time. Therefore Faster-RNNLM places the neural network in shared memory and allows all pthreads to alter the parameters within it without synchronization. With this effect we see that speedup increases nearly linearly, with a constant factor, until the number of hardware threads runs out. A graph of this effect is shown below.



Since the Xeon processors on the Latedays nodes have 12 hardware cores with hyperthreading, altogether there are 24 threads available. So we see a nearly linear speedup up until 24 threads are used, with a small speedup up to 32 threads, and a drop afterwards. The drop is due to the overhead of creating and destroying pthreads, splitting up the data into a

granularity that is too fine, and interprocess communication, since each thread will be reading and writing to the neural network simultaneously. In addition, there is a sequential portion in the timing loop where a single thread calculated the error after the iteration, and this portion causes the factor to be less than 1 for the linear speedup.

We experimented with a different format for reading the input text file when we were attempting to run the process on the Xeon Phi, so we compared the changes when run on the CPU as well. Our changes involved reading the file into a C++ stringstream shared between the threads and having each thread read from that, as well as reading the file into a string and having each thread read from that. These changes resulted in approximately a 20% slowdown, since their optimizations for parsing the text file itself into sentences and words were faster than those in the C++ standard library.

V. Lessons Learned

a. Xeon Phi

Although we had developed code for the Phi before this project, we had not anticipated both the difficulty of implementing existing algorithms onto the Phi and the lack of resources available to aid in porting code to the Phi. It became clear that most users of the Phi do so by implementing their code from scratch, built specifically for use on the Phi. There seems to be a negative feedback loop of not many users developing for the Phi, and Intel not making it easy to develop for the Phi. A great example of this is Intel not keeping their public Torch library up to date with the rest of the major Torch packages that researchers would use. In addition, data needing to be bitwise copyable from a host to the Phi makes it extremely difficult to alter existing code to run on the Phi.

b. Char-RNN vs. Faster-RNN

It became clear that it makes a huge difference which computing architecture a developer is focused on. Since Torch is built for high performance scientific computing on CUDA, Char-RNN performed beautifully on the GPU and scaled very well as available resources were increased. However, when run on the CPU, its performance was very disappointing. If the Char-RNN package itself was developed to more efficiently parallelize training, or if Torch had better adaptability to perform linear algebra without multithreading, perhaps this issue would have been alleviated.

VI. Summary

a. Conclusions

i. Xeon Phi

While we initially thought that running existing code built for regular x86 CPUs will require minor modifications to support offloading code and data to the Phi, we realized that this was not the case. Running on the Phi requires code to be written specifically for the Phi from

the ground up. Char-RNN did not run on the Phi because we were unable to find a suitable version of Torch that successfully runs on the Phi. While there is ongoing work by Intel and third parties like Colfax Research, the repositories online do not have these libraries publicly available.

Even code written in regular C++, like Faster-RNNLM, is not code-compatible with the Phi. The major restriction here is the fact that C++ classes and objects are not "bitwise copyable" to the Phi. To make an RNN run successfully on the Phi, the entire neural network framework along with the BLAS packages need to be rebuilt using much simpler data types.

ii. Char-RNN

We extensively tested the effect of using data parallelism in Char-RNN to better utilize GPU resources. Since data parallelism is weak scaling, and the size of even the smallest data sets has enough data to occupy all the processing elements in our system, this provided a near-linear speed-up for the GPU up to a batch_size of 250-300 (where batch_size is the number of batches being processed in parallel). Beyond this, we see diminishing benefits of increasing batch_size, indicating that the GPU is now oversubscribed.

We expected a similar speed-up for CPUs as well. However, we found upon analysis that the CPU implementation is not well optimized to take advantage of the cache. We saw significant slow-downs as we increased the number of threads. Analyzing the system with perf showed large increases in cache miss rate when the number of threads increased, showing that there is a large amount of optimization that could be performed in order to make Char-RNN more efficient on a CPU.

iii. Faster-RNNLM

It was interesting to note how much more efficient Faster-RNNLM was than Char-RNN on a CPU, even though the code was less sophisticated, purely because Faster-RNNLM was written specifically for a CPU. We saw effects that did not surprise us; namely that speedup was linear until the number of physical cores were exhausted. We saw areas of improvement for this problem, that the neural network and the input training file could be parsed and structured more efficiently to prevent cache invalidation and simultaneous reading and writing to the same memory location.

b. Future Work

i. Xeon Phi

The main goal we would like to achieve in the future would be the same as our original goal in the project: to implement an RNN on the Xeon Phi. We believe that the Xeon Phi could be an extremely useful research tool in that it is able to run x86 code in a highly parallelized environment, as compared to a GPU, where the code must be specifically designed in a domain-specific language, costing many hours in development and optimization. With the Phi, we would be able to use C++ standard library methods to quickly develop and optimize methods for reading and processing training data for an RNN. In addition, we believe that in certain parallelization of an RNN, a Phi could be more optimal than a CPU. When parallelizing over

training data, a balance must be struck on the number of parallel threads training the network, since too many threads will lead to poor convergence or too much synchronization, and too few threads is an inefficient use of hardware. In observing the performance of CPUs and GPUs, we believe that correctly optimized code on the Phi could outperform both in certain cases involving small neural networks and medium-sized training data. Our path for achieving this goal would be to continue trying to use OpenCL as the method for running Torch on the Phi, as there seemed to be the most number of proven cases for this working, and getting OpenCL to run on the Phi seemed to have the fewest roadblocks, namely just upgrading the OS version of the Latedays node. The other alternative mentioned in Intel documentation for the Phi is to use Cilk shared classes that are accessible from the Phi and port existing C++ code to the Phi.

ii. RNNs

The second goal we would like to continue striving for is to study RNNs with model parallelism instead of data parallelism. It would be very interesting to see the comparison of an RNN with model parallelism and an RNN with data parallelism on a CPU, especially the Xeon cores on the Latedays cluster that have the capacity to run up to 24 threads with hyperthreading. Since the performance of Faster-RNNLM dropped off once the cores started hyperthreading, we would like to see if a similar effect occurs under a model parallelism architecture.

c. Final Notes

The credit should be distributed 50/50. Taylor and Anirban contributed equally to the code, the analysis, and the reports. Also, as a note, please refer to the "master" branch on Github (https://github.com/tclarke13/Parallel_Final_Project/) for the basics of our code, although we had many changes and many experiments outside this branch, and some not even reflected on Github as we did not wish to commit them.

VII. References

1. http://karpathy.github.io/2015/05/21/rnn-effectiveness/
2. https://arxiv.org/pdf/1404.5997v2.pdf
3. https://github.com/karpathy/char-rnn
4. https://github.com/yandex/faster-rnnlm
5. http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-processor-product-brief.html
6. http://torch.ch/
7. https://github.com/hughperkins/distro-cl
8. https://github.com/ColfaxResearch/Torch-distro
9. https://github.com/intel/torch
10. https://software.intel.com/sites/default/files/managed/05/ba/heterogeneous-programming-model.pdf
11. https://people.eecs.berkeley.edu/~brecht/papers/hogwildTR.pdf