When the GPU is not the Optimal Choice for Deep Learning

Kevin Martin
Engineering & Computer Science
Syracuse Univ.
Syracuse, NY USA
kmarti44@syr.edu

***Abstract*—Modern graphical processing units (GPUs) are designed to handle a different type of workflow than central processing units (CPUs). The inherent difference in tasks required has shaped the evolution and construction of each. One side benefit has been that GPUs are often able to process the steps needed for machine learning much better than comparable CPUs. Indeed, this is a well known and often studied comparison. However, in this paper we examine situations, through the lens of machine learning, which are the exact opposite and the CPU dominates the required calculations. This disparity comes down to two major elements: task to be performed and choice of hardware.**

***Index Terms*—Machine learning, deep learning, GPU, CPU, hardware, tensorflow, CUDA**

I. Introduction

While originally designed to handle a very specific task, modern GPUs are now built to accommodate a wider range of responsibilities. This spectrum is much narrower than the breadth of asks the CPU must handle, but the applications are growing. CPUs are still required to manage the entire system in addition to performing incredibly fast calculations, and their design reflects this "jack of all trades" nature. Conversely, a GPU is built to process a specific type of input for which it excels at. Historically, this input was rendering graphical images, but the similarities between rendering and certain types of data processing became starkly apparent a few years ago. Specifically, Nvidia Corporation developed an entire framework (Compute Unified Device Architecture or CUDA [1]) for their own devices that could be easily leveraged by machine learning algorithms. This framework encompasses both a hardware component, in the form of specifically designed processing cores, as well as a software component, in the form of high level functional programming APIs.

The major use case of CUDA is for the efficient manipulation of large blocks of data that can be done in parallel. In contrast to sequential program execution where each instruction is executed one at a time, parallel programming allows for multiple instructions, or "blocks" of code to be executed concurrently. Consider a scenario where a program is designed to provide graphics shading: one program thread draws one vertex (shades one pixel fragment) to provide realistic shadowing effects as described by Nickolls et al. [2]. A GPU can process thousands of threads independently by using data parallelization, the fully rendered image can be observed much quicker than if it were done in a sequential fashion. Similarly, the result of a large matrix multiplication operation (a key ingredient in many machine learning algorithms) can also benefit from this strategy. The CUDA framework allows the designer to exploit these benefits by coupling the software with an appropriate piece of hardware. Together, this design allows for a great speedup in certain types of processing.

Briefly, machine learning is a way to train an algorithm to "learn" by observing many instances of a phenomena and updating outputs accordingly. The spectrum of machine learning algorithms can range from relatively unsophisticated applied statistics (such as a feature classification using the Naive Bayes algorithm [3]) to neural networks that require layers "hidden" from the user. The later creates the divide between traditional machine learning and the more complex deep learning. A basic neural network consists of of processing elements, sometimes called nodes, interconnected with one another. Each node outputs a single signal to as many other nodes as desired. The nodes are organized in groups called layers which allow these signals to meaningfully processed by each next grouping. Finally, a uniform output is presented that represents the relationship between each node and each layer [4]. Deep learning takes this a step further by applying nonlinear processing units that transform the signals from a lower level to a higher level automatically. The result is a "representation-learning" method with multiple levels of representation, moving from lower-level to higher-level features (or inputs) [5].

In order to process data in such a way, lots of computational power is required. Intuitively, one may surmise that a properly designed program on a thoughtfully designed piece of hardware dedicated to a specific operation would outperform a more general purpose alternative. Throughout this paper, we observe numerous examples which prove this empirically. However, we postulate that this occurrence is not necessarily a given, and that there are legitimate instances where a CPU will outperform a GPU even under optimal circumstances.

II. Related Work

The interest in optimizing machine learning models has been effectively studied over time, but also requires continued study as both hardware and software change. The original

CUDA API was originally developed for C/C++ and Fortran. As such, the applications were limited to work in those languages only. Eventually, third party wrappers began to be developed for other languages, allowing the application to be widespread. Increased adoption leads to increased performance, and the CUDA architecture has only improved over time.

Schlegel [6] summarizes the application of deep learning on GPUs very well by providing an outline of how neural networks can take advantage of GPU architecture. The experiments and results recorded are also very insightful, but they were done on hardware that is now quite out of date by today's standards. Li et al. [7] dive a little deeper into one specific area by focusing on a subset of neural networks, the convolutional neural network (CNN). The hardware used here is only slightly more modern, but the analysis of how the CNN algorithm works provides valuable information as to why it is an excellent candidate for the GPU as opposed to the CPU.

In addition to deep learning, more traditional machine learning algorithms have also been optimized to take advantage of GPU architecture. Ligowski et al. [8] successfully applied a novel implementation of the Smith Waterman algorithm to scan databases. The key component here was, once again, parallelization. Cederman et al. [9] and Kuang and Zhao [10] each utilized CUDA architecture for enhanced speedups of the quicksort algorithm and K-Nearest Neighbors (KNN) algorithm, respectively. Finally, Mitchell and Frank [11] were able to to accelerate the the training for the XGBoost algorithm by using similar techniques. The XGBoost algorithm is a gradient boosting technique that is a more advanced version of a decision tree algorithm. Here boosting refers to the ensemble learning technique whereby many models are constructed sequentially, and each new model attempts to correct for deficiencies in the previous iterations. XGBoost is a powerful algorithm and its speedup through CUDA shows the power of this approach.

III. Background

Utilizing CUDA is not as simple as merely invoking its API. Each program instance needs to be configured with not only the proper drivers, but also matching versions of each driver. Additionally, not every graphics card from Nvidia is equipped with CUDA support. Thus to create an optimized environment, much planning is required. For these tests, the choice was made to use consumer-grade or "enthusiast" level hardware. By that we mean components that can be assembled by an average desktop user using readily available parts and standard fitments. We demonstrate that a mid-range CPU can, in some instances, outperform a mid-range GPU even in a properly configured environment.

A. CPU Selection

The current lineup of processors from AMD represent an excellent performance value for the money. Debating one processors merits, or even one company's design decisions, are outside the scope of this paper. We feel that the Ryzen 3600 is an optimal choice to illustrate our objective. With 6 cores and 12 threads ranging in clock speeds from 3.6GHz to 4.2GHz [12], this particular chip objectively outperforms every other processor in every paper referenced, with the exception of one test from Mitchell and Frank [11], which uses two server grade CPUs. However both other configurations used still underperform our Ryzen. It should be noted that this chip retails brand new for $175, which is very affordable considering that just a few years ago this amount of processing power did not exist in a consumer-grade chip.

B. GPU Selection

In selecting a GPU, the choice becomes much more difficult. While there are numerous cards that are built with CUDA cores, Nvidia currently has four enthusiast cards (dubbed the 20-Series GEFORCE RTX cards) in production [13]. The prices range from $400 - $1,200. Generally, these cards are sold more for computer gaming, but still pack impressive amounts of CUDA cores. Most studies opt for a higher tier of card, a professional grade GPU that can range anywhere from $2,500 (Titan RTX) to well over $8,000 for their server-grade Tesla cards.

In our opinion, pairing a capable mid-range CPU against an excellent top tier GPU does not make for a great comparison. One would expect a piece of hardware costing 10-50x times more to easily outperform the CPU. When designing a machine for a specific task, such as deep learning, cost becomes a major factor. As such, we opted for the most budget-friendly option that still contains the most current (as of this writing) chipset from Nvidia: the 2060 Super. With 8GB of GDDR6 memory, a processing speed of between 1.47GHz - 1.75GHz, and 2,176 CUDA cores, this is still a very powerful GPU. In comparison used by previous studies, it also generally outperforms other cards used. Advances in physical chip creation have allowed for many more transistors to be packaged on a single die allowing for greater computational power. Only the Titan X used by Mitchell and Frank [11] outperforms the 2060 Super. At $400, it is still over 2x the price of the Ryzen CPU, but certainly within the same tier of performance.

C. Environment

To make full use of the CUDA architecture, we utilized the Python wrapper Keras [14]. Keras is a user-friendly API that allows for clear and objective code construction in creating a neural network. It runs on top of TensorFlow [15], the Google-created library which allows for interaction with the CUDA framework. By combining appropriate versions of each, as well as Nvidia's current CUDA driver and cuDNN driver (specific to deep learning), we have the required framework. Additional notes are we used Window's 10 Operating System, and configured the system environment variables for the correct PATH option. Finally, these tests were run using Python 3.7 (at the time of this writing, version 3.8 does not have support for this setup).

IV. Techniques Used

As discussed, the biggest advantage of a GPU over a CPU is the efficient processing of parallized instructions. It should be noted that not every program is optimized for this type of execution, and in fact many programs cannot take advantage of this technique at all. As such, we would expect that some programs will run faster on a CPU than a GPU. In looking at various machine learning techniques, we are examine two different ends of the spectrum: those optimized for parallelization, and those that are explicitly sequential in nature.

In order to standardize the comparison and create as fair a comparison as possible, we utilize the same dataset through two different lenses. We look at the "IMDB" dataset [16] that is available through the generic Keras API. We chose it because it requires no additional manipulation or cleaning techniques to become usable. In general, the available datasets within an API's library are often curated by the maintainers and serve as optimal examples for machine learning. In this particular case, the dataset provides approximately 50,000 movie reviews labeled by a binary favorable or unfavorable sentiment. Here the data is preprocessed to allow for each review to be encoded as a sequence of word indexes with integers. Each word within a review is indexed by overall frequency within the global dataset. We can set up two different algorithms to evaluate the dataset, and while the accuracies are interesting, the time to completion is the main focus of the experiments. Both are variations on the artificial neural network (ANN) with multiple hidden layers. An ANN uses multiple inputs and passes the information to the next layer of nodes. Each node processes the signals it receives, and passes that information on to the next layer. The number of layers can vary as needed, but the end result is an output which is the combination of all the previous layers. See Figure 1 for an example of a basic ANN, with two inputs and two hidden layers:

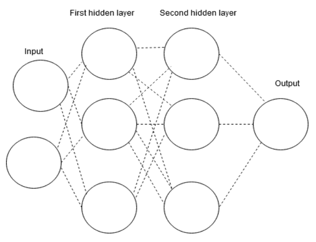

Fig. 1. Basic ANN with two hidden layers

For the GPU-focused algorithm, we look at a convolutional neural network (CNN). The CNN is a specific type of forward-feed ANN that utilizes a "pooling" approach, as seen in Figure 2:

As described by Lopez and Kalita [17], the main difference is in the amount of layers utilized. The convolutions are just a sequence of nonlinear activation functions which are applied to the results of the previous layer. Features are extracted from the original input and pooled together during subsequent

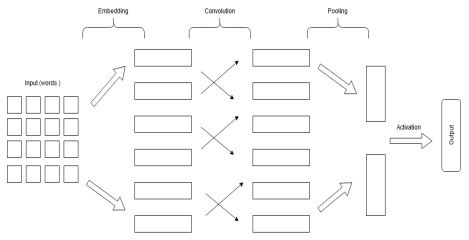

Fig. 2. CNN flow for text processing

layers, which are finally connected for the unified output. The inputs are the sentences which get represented as a matrix, where each row is a token (in this case a single word). It is in the processing of the matrix where the benefits of parallelization can be realized.

As a counterpoint, we look at another viable method for processing the IMDB dataset, the bi-directional long-short term memory (LSTM) model. The LSTM is a variation on the ANN and uses a looping or recurring component on the hidden layer, see Figure 3.

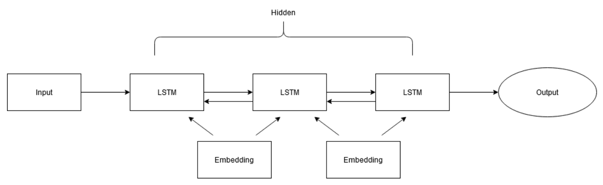

Fig. 3. Bi-directional LSTM flow

This addition of both forward and backward passes allows for better results as we have access to both past and future inputs. The calculations are still carried out in a sequential manner even with this addition. Furthermore, the hidden layer updates are replaced by purpose-built memory cells, as described by Huang et al. [18]. The goal is for the model to focus on long range dependencies in the data. As explained by Ghosh et al. [19], this can be very valuable in text processing, even including contextualization. Importantly, the basic format is sequential in nature (in opposition to the CNN) and thus does not benefit from parallelization. As such, we would expect this model to perform better on a CPU than a GPU because the CPU is more suited to sequential tasks. However, this is by no means a given truth: very easily, one could conceive of a system with a (relatively) weak CPU and a (relatively) strong GPU where the algorithm would still perform better on the GPU despite the lack of parallelization exploitation available. Conversely, the same could be said of a reversed setup and a CNN. We want to illuminate that the CNN can make use of parallelization and thus would favor the GPU's architecture, and the LSTM model relies more on sequential processing which favors the CPU's architecture.

V. Results

Even though each algorithm used is a variation on the ANN, we see vastly different performance results. We look at performance through two lenses: accuracy of the models as well as training time. Training time is the main focus as it will be the benchmark by which we evaluate the relative speeds of each hardware component. However, we want to keep accuracy in mind as well as a reasonableness check. If the models cannot generate accurate results then we do not believe the training is useful, and thus any times become irrelevant. As this dataset has not only been preprocessed but also serves as a learning example, we expect accuracies of at least 75% in each instance. The IMDB dataset was first created and used by Sadeghian and Sharafat [20] for use in a Kaggle competion. The researchers were able to achieve an accuracy of 90.9% under the most optimal of circumstances. We would naturally expect a range of accuracies for our tests as some of the runs are done in very sub-optimal conditions purely to demonstrate the time it takes each model to train and generate its prediction.

To evaluate each scenario, we hold all hyperparameters constant except for the epoch. Each epoch represents how many times the model sees the entire dataset. In general, increasing the amount of epochs increases the accuracy as the model has more instances to observe patterns in the dataset. There is a trade-off, and too many epochs can subject the results to overfitting, which is when the model has learned the dataset too well and is no longer making predictions but rather trying to recreate what it has already observed. Additionally, increasing the epochs creates another trade-off in terms time and computational resources. As such, we can observe the efficiency of each device as the amount of epochs increase.

A. CNN

The CNN performed objectively very well, achieving between 86.48% - 88.48%. The pooling effect appears to be very well suited for this type of text processing. Indeed, with only minor hyperparameter tuning, we were able to achieve similar results to the original Kaggle competition despite that not being our intended goal. We observe that the performance figures between the CPU and GPU are extremely similar, and this is helpful to keep in mind. There is no benefit to choosing one over the other in terms of accuracy. Based on our findings, the decision should instead be left to speed of training.

Additionally, we noted little to no benefit with increased epochs. As seen in Figure 4, both hardware components offered similar fluctuations across training runs. We believe this effect is due to two main things. First, the general effectiveness of the CNN and how it handles this dataset. The CNN is optimized for such tasks, and with adequate batch sizes and inputs, achieve excellent results without the need for multiple training runs. Second, the task at hand is just not that difficult for the algorithm. The patterns appear to be clear and relatively easy for the model to decipher. For our purposes, the accuracies are more than sufficient to justify their usage.

In comparing the speeds of each, we observe significant gains from the GPU which support our initial hypothesis. In Figure 5, the benefit is quite obvious in terms of not only epoch-to-epoch, but also as the number of epochs increase. The GPU scales much more linearly, while the CPU appears

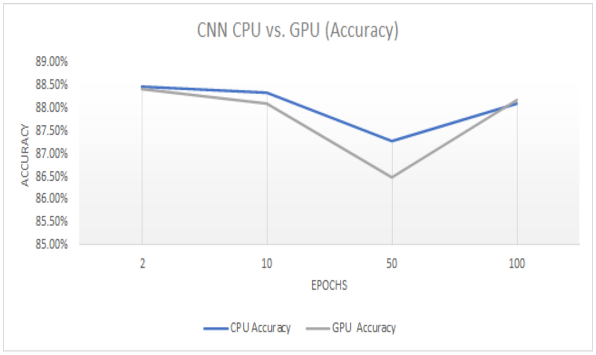

Fig. 4. CNN accuracy results

to be requiring more time as the amount of work increases. Should the need for more epochs be increased, the GPU would continue to be the clear choice, while the CPU would be rendered unusable.

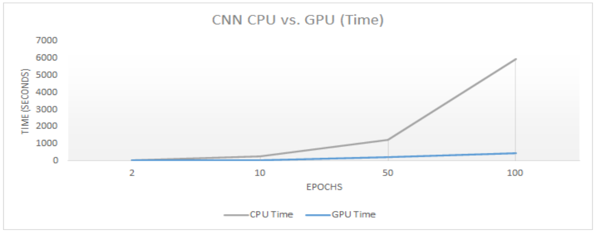

Fig. 5. CNN time results

B. LSTM

The LSTM also performed well, still within the an appropriate and valid range, though certainly less impressively than the CNN. At best, we observe an accuracy of 84.5%, which is still worse than even the lowest CNN result. We believe that nature of the LSTM algorithm, while decent, is simply not the best choice to solve this type of problem. With additional hyperparameter tuning we imagine the results improving, but most likely not to the degree achieved with the CNN. Another interesting observation was that of the decrease in performance as the number of epochs was increased. In Figure 6, the general downward trend can be seen by both the CPU and GPU. Again, as we are not concerned with optimizing accuracies, these results are sufficient.

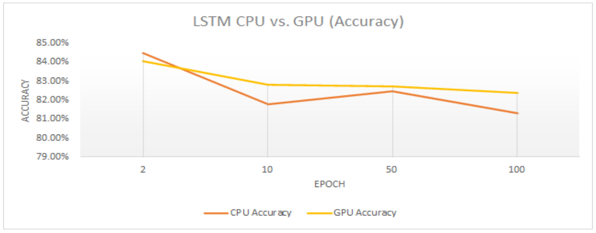

Fig. 6. LSTM accuracy results

As expected, we see the CPU outperforming the GPU on the more sequential-oriented task. As the epochs increase, so does the training time. We note that this effect is more gradual than with the CNN, specifically in the case of the CPU which begins to ramp up quite steeply. The difference on the LSTM is much closer as well, with less time benefit in choosing the more appropriate component. See Figure 7:

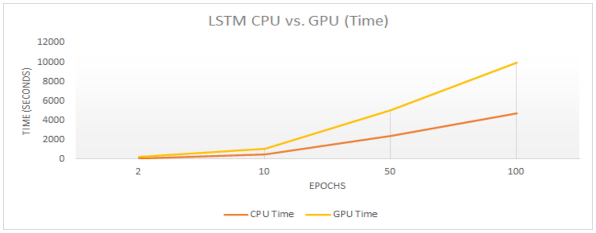

Fig. 7. LSTM accuracy results

C. Overall Comparison

In Figure 8, the results of both time comparisons are presented together. The clear choice to accomplish this task is the CNN on the GPU. Not only did it give near the best results, but in a much shorter time. In a production environment, this allows for quicker time to analyze the results and tune the model. This result is not surprising, and indeed conducive to what the majority of similarities have determined. However, our big takeaway is how poorly the same GPU performed on the LSTM. The performance is poor both in terms of accuracy and speed.

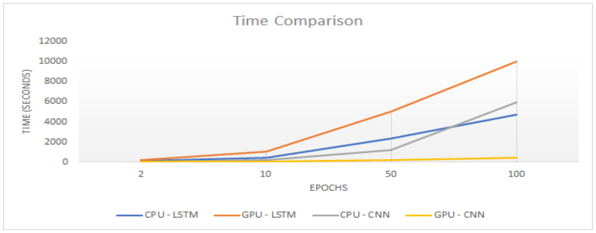

Fig. 8. Overall time results

Interestingly, the CPU performed similarly in both applications. Given the high variety of requests and tasks the CPU must accomplish, it is designed to be "good" in all circumstances. Conversely, the GPU is designed to accomplish only a very specific task, but accomplish it very well. Performing matrix calculations using parallelization is one of those tasks and the effects are quite apparent.

VI. Conclusions

In considering how to train a model to tackle an advanced machine learning problem, the generally accepted best approach would be to utilize a powerful GPU. Through our tests, at the consumer-grade level, we have shown that this is true with one large caveat: the algorithm must be optimized for this type of hardware architecture. A capable CPU will vastly outperform a capable GPU on a task that is largely sequential in nature. Conversely, the same GPU will outperform the CPU in a task that can be effectively paralleled. The gains of utilizing the GPU in this case will be far greater than utilizing a CPU over a GPU in a sequential task. Thus our conclusion is that, for a machine learning algorithm, the impetus to exploit parallel computing is very high. Furthermore, this is consistent with the work that has been done before on the subject. In instances where parallelization is not an option, as in the case of a sequential algorithm, it may be more beneficial to utilize the CPU. There exits a multitude of variables that need to be considered, but if the CPU and GPU in the system are of the same class, there in fact are scenarios in which the CPU will be the optimal choice.

References

[1] [Online]. Available: https://developer.nvidia.com/cuda-zone
[2] B. Nickolls and S. Garland, "Scalable parallel programming with cuda," *ACM Queue*, 2008.
[3] I. Rish, "An empirical study of the naive bayes classifier," *IJCAI 2001*, 2001.
[4] R. Hecht-Nielsen, "Theory of the backpropagation neural network," *Elsevier*, 1992.
[5] T. Zhang and Z. Han, "From machine learning to deep learning: progress in machine intelligence for rational drug discovery," *Drug Discovery Today*, September 2017.
[6] D. Schlegel, "Deep machine learning on gpus," *Seminar Talk*, December 2015.
[7] Y. Li *et al.*, "Optimizing memory efficiency for deep convulutional neural networks on gpus," *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, November 2016.
[8] R. Ligowski, "An efficient implementation of smith waterman algorithm on gpu using cuda, for massively parallel scanning of sequence databases," *2009 IEEE International Symposium on Parallel Distributed Processing*, July 2009.
[9] D. Cederman and P. Tsigas, "Gpu-quicksort: a practical quicksort algorithm for graphics processors," *ACM Journal of Experimental Algorithmics*, January 2010.
[10] Q. Kuang and L. Zhao, "A practical gpu based knn algorithm," *Proceedings of the Second Symposium International Computer Science and Computational Technology*, December 2009.
[11] R. Mitchell and E. Frank, "Accelerating the xgboost algorithm using gpu computing," *PeerJ Computer Science*, July 2017.
[12] [Online]. Available: https://www.amd.com/en/products/cpu/amd-ryzen-5-3600
[13] [Online]. Available: https://www.nvidia.com/en-us/geforce/20-series/
[14] [Online]. Available: https://keras.io/
[15] [Online]. Available: https://www.tensorflow.org/
[16] [Online]. Available: https://keras.io/api/datasets/imdb/
[17] M. M. Lopez and J. Kalita, "Deep learning applied to nlp," *arXiv:1703.03091*, March 2017.
[18] W. X. Zhiheng Huant and K. Yu, "Bidrectional lstm-crf models sequence tagging," *arXiv:1508.01991*, August 2015.
[19] S. Ghosh *et al.*, "Contextual lstm (clstm) modesl for large scale nlp tasks," *arXiv:1602.06291*, February 2016.
[20] A. R. S. Amir Sadeghian, "Bag of words meets bags of popcorn," *Stanford University*.