Exploring the power of GPU's for training Polyglot language models

Vivek Kulkarni, Rami Al-Rfou', Bryan Perozzi, Steven Skiena

Department of Computer Science
Stony Brook University
{bperozzi,ralrfou,vvkulkarni,skiena}@cs.stonybrook.edu

Abstract. One of the major research trends currently is the evolution of heterogeneous parallel computing. GP-GPU computing is being widely used and several applications have been designed to exploit the massive parallelism that GP-GPU's have to offer. While GPU's have always been widely used in areas of computer vision for image processing, little has been done to investigate whether the massive parallelism provided by GP-GPU's can be utilized effectively for Natural Language Processing(NLP) tasks. In this work, we investigate and explore the power of GP-GPU's in the task of learning language models. More specifically, we investigate the performance of training Polyglot language models[1] using deep belief neural networks. We evaluate the performance of training the model on the GPU and present optimizations that boost the performance on the GPU.One of the key optimizations, we propose increases the performance of a function involved in calculating and updating the gradient by approximately 50 times on the GPU for sufficiently large batch sizes. We show that with the above optimizations, the GP-GPU's performance on the task increases by factor of approximately 3-4. The optimizations we made are generic Theano[2] optimizations and hence potentially boost the performance of other models which rely on these operations.We also show that these optimizations result in the GPU's performance at this task being now comparable to that on the CPU. We conclude by presenting a thorough evaluation of the applicability of GP-GPU's for this task and highlight the factors limiting the performance of training a Polyglot model on the GPU.

1 Introduction

GPU computing has been of significant interest to the research community. This has led to the evolution of heterogeneous parallel computing with the realization that clock speeds on CPU's have reached their limit. Heterogeneous Parallel Computing offers a solution to this problem by offloading computation that can be massively parallelized on the GPU's and running serial computation on the CPUs. Thus GP-GPU applications are widely used to solve several computational problems. For example, BarraCUDA is a software that uses GPU's to speed up the alignment of short sequence reads to a particular location on some reference genome[3]. GPU's have always been extensively used in the field of

computer vision for image processing. To illustrate, OpenVidea is a system that implements several computer vision algorithms on the GPU[4].It is to be noted, that most images are represented as dense matrices, and image processing algorithms render themselves to be effectively parallelized as they tend to operate on stencils(a fixed template of pixels). Since GPU's are particularly suited for vector computations, image processing applications typically are designed to utilize the massive parallelism offered by the GPU's.

In contrast in the field of Natural Language processing, the models are typically sparse. This implies that language models have a sparse representation. Since the model is sparse, the computation does not admit to being effectively accomplished on the GPU. However recent advances in machine learning and natural language processing have resulted in the development of new language models that learn representations[5][6]. These representations which are also termed *embeddings* are a dense representation. One popular technique of learning word representations is to use deep belief networks. Deep Belief networks are neural networks which tend to have a large number of hidden layers. Training deep belief networks requires massive amounts of training examples and thus requires massive computation. Deep belief networks used for solving computer vision tasks are most efficiently trained on the GPU[7]. In this work, we investigate the performance of training such a language model(learnt by deep belief network) on the GPU. The language model we used for learning word representaions is very similar to the language model described in the *SENNA* system[8]. We call our language model Polyglot. Polyglot allows one to learn word representations from massive amounts of unannotated text. We have used Polyglot to learn word embeddings for more than 100 different languages[1]. A detailed description of the Polyglot project is available in [1].At a high level, we measure the performance of training a Polyglot model in terms of the number of training examples processed per second. We evaluate the performance in terms of the speed-up achieved on the GPU and compare it against the CPU. We next briefly describe our experimental setup and describe our research methodology.

2 Experimental Setup

We run all our experiments on the GPU on GEForce GT 570 . This GPU has 480 cores, with a processor clock speed of 1464MHz and a memory clock speed of 1900MHz. GPU's in the GEForce family are particularly suited for research applications. To train the language model, we use a well-known library called *Theano*[2]. Theano is a symbolic computation library that is particularly suited for machine learning applications. It allows for developers to succinctly specify how the parameters of the model are to be calculated. Using Theano to learn a model, allows the developer to implement a machine learning algorithm easily as the complexity involved in explicitly calculating the gradients of the loss function is abstracted out by Theano. Theano has functionality built in to calculate gradients and perform mathematical computation easily. Theano also

[1] Available: http://bit.ly/embeddings

transparently offloads computation on to the GPU if required. Since training a model, significantly involves computation required to learn a set of parameters that minimize a loss function, it is imperative that the functions implemented in Theano are efficient. This involves performing both macro level optimizations in optimizing the computational graph and micro level optimizations that target specific functions. Having described our experimental setup, we now describe at a high level, our research methodology in identifying bottle necks and optimizing these bottle necks.

3 Research methodology

We follow the below standard procedure to analyze and evaluate the performance bottlenecks:

1. Note the number of training examples processed per second on the CPU and GPU to establish a baseline.
2. Use a profiler to identify top hot spots.
3. Focus on the top hot spots and investigate how they can be optimized
   (a) Identify any computational graph optimizations possible to reduce computation
   (b) Optimize function calls by investigating how parallelism can be boosted.
   (c) Micro-optimize function calls
   (d) Unit Test the changes to ensure correctness
4. Repeat Steps 1 to 3 if required.

4 Results

4.1 Baseline

In this section, we present the baseline numbers for the GPU and the CPU. On the CPU, the mean training rate was 5512.6 examples/second($\sigma = 30.315$). On the GPU, the mean training rate was 1265.8 examples/second($\sigma = 20.604$).

The goal is to investigate the performance on the GPU and evaluate the bottle-necks involved. On scanning the logs, we note that the percentage of total time approximately spent on Theano processing was 96%.

This implies that it would be fruitful to analyze and focus on optimizations in Theano to help boost performance. In the next section, we analyze the performance of Theano using the built in Theano Profiler.

4.2 Profiling Theano

We profiled Theano to get some insight into what the performance hot spots are, so that our efforts could be channeled into optimizing those hot spots. We show the time taken per call and the fraction of time spent for these functions

Table 1: Top 3 Hot spots in Theano

Theano Function	*Fraction of time spent*	*Time per call to function*
GpuAdvancedIncSubtensor1	81.7%	4.60×10^{-3}s
GpuElemwise	9.2%	6.93×10^{-5}s
GpuAlloc	1.7%	1.91×10^{-4}s

in the Table 1 .We note that there is 1 major hot spot, namely *GpuAdvanced-IncSubTensor1*.

We thus narrow down our goal to optimizing the function *GpuAdvanced-IncSubTensor1*. This function typically performs an operation called *advanced-indexing* which is an operation that is typically performed while calculating the gradient of parameter vector and updating them. In order to optimize this, it is crucial to understand what the operation of *advanced-indexing* does.

This operation operates takes as input 3 parameters: W and Y which are matrices and a vector I. The vector I refers to (indexes) rows in W. Given a row of W indexed by I, this operation adds the corresponding row of Y to it to form the output. This is done for each row indexed by I.

Having understood the operation that we are targeting to optimize, we now outline in the next section, the optimizations we investigated and present how the performance of this function improves with these optimizations.

4.3 Optimizing advanced indexing

To enable quick testing of advanced indexing, we wrote a standalone script that invokes the advanced indexing operation. This enables us to quickly test our fixes and ensure their correctness.

On examining the code for the advanced indexing, the following was noted:

1. The implementation of advanced indexing was done in Python and can be slower than an implementation in C. Hence it would be beneficial to rewrite the implementation in C to boost performance.
2. The code was not highly parallelized and had a low degree of parallelism. So we decided to not only write an implementation of advanced indexing in C, but parallelize it as well. This was done by writing a CUDA kernel which would perform the advanced indexing in parallel. More specifically instead of indexing each row sequentially, each row is indexed in parallel, and for each row, each cell in the row is added in parallel. This greatly boosts the parallelism of the algorithm.
3. Another micro-optimization that was tried was to also make the operation in-place at the cost of an extra memory copy. However this was shown to give diminishing benefits.

With these optimizations, the mean time taken for indexing 1000 rows is 3.6612 seconds($\sigma = 0.14116$) compared to the baseline where the mean time for indexing

1000 rows was 207.59 seconds ($\sigma = 2.9652$). We also note specifically that the time per function call to advanced indexing has reduced by a factor of 50.

The above speedup thus results in Advanced Indexing no longer being the bottle-neck. In the training of the model, we do not index as many rows(as the context size and batch sizes are smaller) and hence the speed up in training is expected to be lower. One of the reasons for having small batch sizes is that the model converges faster. We in fact noted that increasing that batch size to 500 results in a much slower convergence rate of the model(as we will present in Section 4.6).

4.4 Speedup in training rate

The mean rate of training is now 3742 examples/s($\sigma = 32.6496$). We thus note that we have achieved a reasonable speed up of $3 - 4$ times on the GPU with our optimizations and the performance on the GPU is comparable to that on the CPU. We also note that advanced indexing is no longer a major bottleneck in the training task. On performing these optimizations, it is imperative to do a further analysis on whether further optimizations are worth the "bang for the buck". This requires us to analyze exactly what is limiting the speedup. Thus in the next section we present our analysis on what is limiting the speed up on the GPU and highlight our findings below and outline the next steps.

4.5 Analysis of limits on training performance on GPU

We profiled the entire application(after we included all the optimizations above) and analyzed the profile log using NVIDIA Profiler(nvprof). From the profile, we extracted the following metrics:

1. *Compute Utilization:* This is the fraction of total time spent executing on the GPU. We would like the compute utilization to be high. If this ratio is small, this indicates that most of the time, the GPU's are idle. For our task, we note that the Compute Utilization is 7.4% which is low
2. *Compute to Memory Op Ratio:*Out of the time spent executing on the GPU, what fraction of time was spent doing computation to amount of time spent in transferring data to and from device is the Compute to Op Ratio. This ratio should be high and at least $10 : 1$. We noted that this metric is 66.72 which is high.
3. We also note the top 2 kernels as follows:
   (a) *Composite Kernel*:This kernel performs an element wise operation on a C array which is contiguous in memory and is highly optimized with less scope for optimization.
   (b) *copy_kernel*: This is a kernel which is a part of the BLAS Library.
   (c) Also the above kernels are not expensive as they don't have expensive operations like exponentiation etc and hence are not bottle-necks.

We thus conclude that the performance of training on GPU is limited by the low compute utilization which implies that the GPU cores are mostly idle. The low compute utilization is a fallout of the model, as we are unable to fully utilize the GPU with our current implementation. One technique of improving compute utilization in this task would be to increase the batch size of examples which implies that each batch would contain more examples and thus increase the computation on the GPU. In the next section, we present our results on this task for increasing batch sizes.

4.6 Effect of Batch Size on training and convergence rates

Currently, the batch size is set to 16. We decided to try a range of increasing batch sizes from 16 to 512 and measured the training rate and the time taken by the model to converge to an error less than 0.05. We make the following observations(see Figure 1):

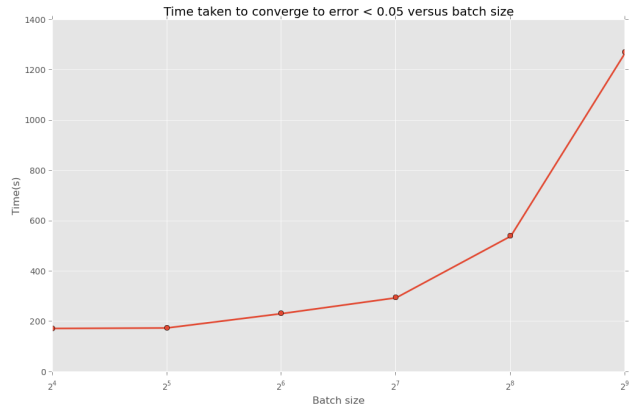
1. The training rate does increase as we increase the batch size.
2. The time taken to conerge to a given error grows linearly (note the log scale on the X-axis) as we increase the batch size

The observation that the convergence rate decreases as we increase the batch size can be explained by the observations made by [9] that batch training is generally inefficient as compared to online training. We note that by increasing the batch size, the updates to weights accumulate and result in larger updates. This results in the gradient descent taking unreasonably large steps thus possibly overshooting the local minima on the error surface. We thus conclude by noting that increasing the batch size is not an effective strategy to speed up training as the model converges more slowly. In the next section, we highlight future research directions that could be investigated to speed up training.

5 Future Work

One area which could be investigated is to use the distributed algorithms for calculating gradients(using gradient descent) outlined by Jeffrey Dean et.al in [10] in the model and evaluate its performance. These algorithms update the weight vector in a distributed fashion (with updates not being synchronized). It is claimed that distributed stochastic descent performs reasonably well for the models learnt. It would also be interesting to investigate if performance of training other models can be optimized on the GPU.It has been shown in [11] that Hellinger PCA can be used to learn word representations and perform competitively against word representations learnt by other models. It would be interesting to investigate whether this is amenable to good parallelization on the GPU. A second direction would be to evaluate if we could effectively use GPU's for other per-processing tasks like annotating the Web etc.

(a) Effect of batch size on training rate.



(b) Effect of batch size on convergence rate

Fig. 1: Effects of batch size on training and convergence rates.

6 Conclusion

In this section, we briefly summarize our contributions below:

1. We were able to significantly optimize the operation of advanced indexing on the GPU in Theano.
2. We showed that this boosted the performance of training on the GPU by factor of 3-4 times and is comparable to the performance on CPU
3. We were able to analyze exactly what the limiting factors for training performance were on the GPU and showed that this performance was limited by low compute utilization.

4. We analyzed the effect of increasing batch size on the training speed and the convergence rate and concluded that the model converges slower for larger batch sizes even though we obtain a good speedup in training.
5. We also contributed back our optimizations to the Open source community so that these optimizations would benefit other members of the research community as well.

7 Acknowledgements

We thank the Theano development team for helping out on code reviews for the patches submitted to Theano.

Bibliography

[1] Rami Al-Rfou', Bryan Perozzi, and Steven Skiena. Polyglot: Distributed word representations for multilingual nlp. In *Proceedings of the Seventeenth Conference on Computational Natural Language Learning*, pages 183–192, Sofia, Bulgaria, August 2013. Association for Computational Linguistics. URL http://www.aclweb.org/anthology/W13-3520.

[2] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*, June 2010. Oral Presentation.

[3] The barracuda project. http://seqbarracuda.sourceforge.net/index.html.

[4] James Fung and Steve Mann. Openvidia: Parallel gpu computer vision. In *Proceedings of the 13th Annual ACM International Conference on Multimedia*, MULTIMEDIA '05, pages 849–852, New York, NY, USA, 2005. ACM. ISBN 1-59593-044-2. doi: 10.1145/1101149.1101334. URL http://doi.acm.org/10.1145/1101149.1101334.

[5] Geoffrey E. Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural Comput.*, 18(7):1527–1554, July 2006. ISSN 0899-7667. doi: 10.1162/neco.2006.18.7.1527. URL http://dx.doi.org/10.1162/neco.2006.18.7.1527.

[6] G. E. Hinton and R. R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, 2006. doi: 10.1126/science.1127647. URL http://www.sciencemag.org/content/313/5786/504.abstract.

[7] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C.J.C. Burges, L. Bottou, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[8] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. Natural language processing (almost) from scratch. *Journal of Machine Learning Research*, 12:2493–2537, Aug 2011. URL http://leon.bottou.org/papers/collobert-2011.

[9] D. Randall Wilson and Tony R. Martinez. The general inefficiency of batch training for gradient descent learning. *Neural Netw.*, 16(10):1429–1451, December 2003. ISSN 0893-6080. doi: 10.1016/S0893-6080(03)00138-2. URL http://dx.doi.org/10.1016/S0893-6080(03)00138-2.

[10] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc Le, Mark Mao, Marc'Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Ng. Large scale distributed deep networks. In P. Bartlett, F.C.N. Pereira, C.J.C. Burges, L. Bottou, and K.Q. Weinberger,

editors, *Advances in Neural Information Processing Systems 25*, pages 1232–1240. 2012. URL http://books.nips.cc/papers/files/nips25/NIPS2012_0598.pdf.

[11] Rémi Lebret and Ronan Lebret. Word emdeddings through hellinger pca. *CoRR*, abs/1312.5542, 2013.