Batched Shift Reduce Parsing with Lists of Vectors on CUDA

Andrew Drozdov
apd283@nyu.edu

Abstract

Shift Reduce Parsing is a common algorithm used in compilers and natural language processing, and can be used to compose a sequence of fixed-length vectors into a single vector of equal length. Previous versions are implemented using predetermined computational graphs that trade excessive memory and computation to minimize transfers of memory from the device to the host. In this paper, I present a version of Shift Reduce Parsing that uses only the necessary memory and computation, show that the memory transfer is insignificant when implemented properly, and analyze how the sequence of transitions can effect computation when running a batched version of this algorithm, as is common in deep learning programs.

Introduction

Shift reduce parsing is a way to consume trees as a sequence. This would be a good algorithm if we wanted to convert a tree into a sequence. When running Recursive Neural Nets, this is a desire that we have. RNNs like other deep learning algorithms rely heavily on matrix operations including outer products, elementwise operations, and various transformations. In many cases, these operations are parallel, even embarassingly parallel, which is why using CUDA is beneficial. Although CUDA is beneficial, sometimes the matrices are small. This is common, and one of the reasons that deep learning algorithms tend to "batch" iterations by running these operations across many examples simultaneously. It is usually straightforward to batch when all your inputs have the same dimensionality. In natural language, batching examples tends to be more complicated when your input are sentences of varying length. In the case of Shift Reduce Parsing, the problem is exacerbated by the fact examples may have varying tree structure even if they are the same length. In the following sections we will discuss an existing implementation of SR Parsing called SPINN, the tradeoffs that were made in this implementation, and then present the details of my new implementation of SPINN.

1 Background Information

SPINN was introduced by Bowman et al. as a faster way to use Recursive Neural Networks [Bow+16]. It demonstrated you can reach over a 25x speed improvement over traditional Recursive Neural Network implementations by using Shift Reduce Parsing. The most approachable existing open source implementation of SPINN is known as "fat stack". Written in Theano, it treats the buffer and stack as fixed length tensors. When using Theano, this sort of implementation makes a lot of sense. Theano does not have an abstraction for manipulating lists because this does not fit into the paradigm of a "computational graph". Similarly, performing a different set of computations for every example in a batch does not fit either. There are at least two similar frameworks to Theano that do allow for the type of operations that we need. One is Chainer, which like Theano is implemented in Python, but is fairly new and still has some kinks. Another is Torch, which is a seasoned library. In this project, I use Torch two implement two variations of SPINN that vary greatly from the existing one in Theano.

29th Conference on Neural Information Processing Systems (NIPS 2016), Barcelona, Spain.

As confirmation that implementing SPINN efficiently is a non-trivial task, DeepMind recently released a paper that says it was difficult to create an efficient and batched version of SPINN, and confirm this notion after communicating through email [Yog+16].

2 Proposed Solution

There are two primary hurdles in the Theano implementation of SPINN. The first is that it computes a Shift and Reduce at every time step even though clearly only one of these is needed. This is even more subtly wasteful in that when there are sentences of varying length, I compute a Shift and Reduce when neither is needed. Part of my solution will involve computing dynamically sized Reduces, and only performing the necessary Shifts and Reduces for each time step, avoiding any action for transition sequences that have been padded.

The second is that it uses a fixed size Stack and Buffer that are abstracted as matrices. When running on the GPU, it can parallelize updates of these data structures, but storing the Stack and Buffer this way is wasteful from a memory and computation perspective. The other part of the solution will involve using proper array like objects for the Stack and Buffer.

I also evaluate two auxiliary techniques that attempt to reduce overall kernel launches over the course of the model's execution.

The first is attempt to decrease the variance in sequence length within a batch. This can be done by sorting your batches by sentence length. This is straightforward as each example is a single sentence, but for sentence pairs, using a function of both sentence lengths (such as the peano function) could prove useful.

The second attempt maximizes the size of each batched Reduce by seeking through the transitions of each example until a Reduce transition is reached.

3 Experimental Setup

Experiments are run with various parameters on the CPU and GPU. For the CPU, experiments are run on a 2 GHz Intel Core i7 with 8 cores and 8 GB of RAM. I use the default install of torch with OpenBLAS already installed and the default number of threads. For the GPU, all experiments are run on a single Maxwell Titan Black.

The experiments are meant to simulate a batched forward pass with synthetic data. I fix the number of total sentences to 5k, and randomly choose a sequence length by sampling from a Gaussian distribution with mean 13 and standard deviation 3, with a minimum of 5 and maximum of 25. These are similar to the sentence lengths that are seen in the SNLI dataset [Bow+15]. An example is a pair of sentence and its associated transitions.

A sentence of length N has $N * 2 - 1$ transitions. I sample transitions uniformly, ensuring a valid sequence. Examples are grouped into batches, with each experiment specifying its batch size. Each batch is a matrix, so sentences and transitions are padded on the right to a length of 50 in order to align.

One of the questions I'd like to answer is how does batch size effect performance of the program? I expect that it should be roughly unchanged on CPU, but that larger batch sizes should improve performance on the GPU with diminishing returns.

There are 5 batch sizes used: 1, 4, 32, 128, 512. Note than in real machine learning applications, using too large batch sizes can degrade the accuracy of your model if learning with stochastic gradient descent. Gradients are typically averaged during backpropogation, and large batch sizes can result in a too "smooth" gradient. The variance in sentence length within a batch might effect the number of kernel launches.

A simple technique to reduce the variance is to sort the entire dataset by sentence length, then create batches sequentially. A slight alteration that assures batches in each epoch are different is too first group the dataset into buckets before sorting, then sequentially create batches within each bucket. This is not a necessary step when interested in speed, but I still do this to simulate a real environment. The effect on the results is negligible.

It's common to "embed" words into fixed size vectors. Before being passed into the various SPINN implementation, each word is embedded into a vector of size 600. The Reduce operation in SPINN is the parametrized component of SPINN. In this setup, I use a vanilla TreeRNN, but it's worth noting that SPINN uses a TreeLSTM that performs at least 2x as much computation. The equation for the vanilla RNN is as follows.

$$reduce(left, right) = W_L(left) + W_R(right)$$

Where W_L, W_R represent the weights in what would be an operational Neural Network.

4 Results & Analysis

There are 3 different variants of SPINN. They've been labeled as "Pure" (Matrix representations), "SPINN" (list representations), and "Greedy" (lists with opportunistic Shifts/Skips). In addition there are two batching schemes "Random" (nothing clever) and "Bucket" (create batches using sentence length).

At each time step one batch is consumed. I keep track of only the time that the step spends in the core part of SPINN, and ignore the time taken to generate synthetic data and to embed the example. Personal experience indicates that the time spent embedding is negligible.

By aggregating the time from all the time steps, we get the total time spent in SPINN. I also inspect the total number of Reduces. Profiling each batch size for a single time step in nvprof confirmed that the GPU spends the majority of its time in this operation.

Plots of the results are shown in Figures 1, 2 and 3. Bucketed batching has been shown except for in Figure 3.

4.1 Total Time

Figure 1 shows time in seconds to process all examples in SPINN. For all cases except CPU Pure, increasing the batch size gives a speed improvement. I did not experiment with batches larger than size 512, but it looks like performance has mostly plateaued around this size.

4.2 Total Reduces

Figure 2 shows that we have a clear relationship between number of reduce operations and total time.

4.3 Random vs. Bucketed Batching

Figure 3 shows how a combination of bucketed batching and greedy SPINN give the best performance.

5 Other Notes

The original codebase for this project was written in Chainer as this work is inspired by an implementation of SPINN I have done in Chainer. For various reasons, small neural networks do not benefit greatly from using the GPU. Some of these are innate to doing operations on small amounts of data such as latency becoming the bottleneck when there are frequent memory transfers and low utilization within each kernel. These are briefly alluded to in this github issue [1]. For this reason, I rewrote the experiments using Torch, although if interested the Chainer experiments are available in the same github repo as the Torch code.

6 Conclusion

Although using matrix representations of your data is desirable when you can parallelize matrix operations on the GPU, it's certainly worth attempting a solution that could use lists, especially if the of each element in this list would be much smaller than the matrix representation.

[1] https://github.com/pfnet/chainer/issues/1771

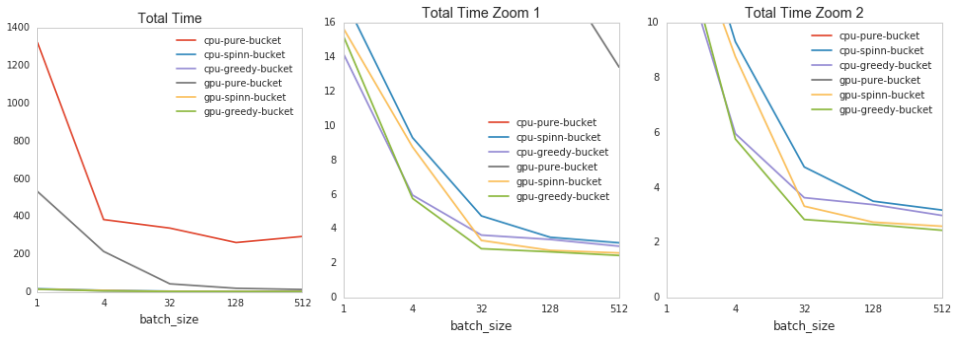

Figure 1: Total time in seconds to complete 5000 examples with varying model size.

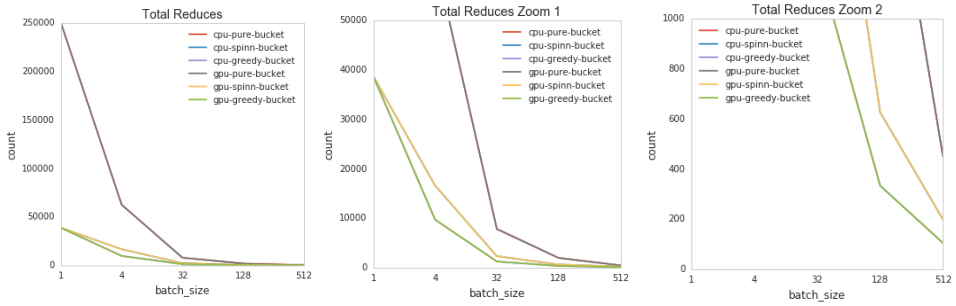

Figure 2: Total number of reduce operations in varying models and batch sizes.

I showed that version of SPINN using lists is much faster than an alternative version that uses matrices. By being opportunistic about Shifting and Skipping transitions, the list version becomes even faster. This improvement in speed likely comes from the decrease in total Reduce operations. In addition, by being clever about composing batches, we decrease the number of Reduces even further giving another small speed bump.

It's unclear which other applications would benefit from list representations, although Neural Turing Machines and Dynamic Memory Networks seem to be likely candidates because like SPINN they rely on data structures that are only partially modified at each time step.

Being careful about batch arrangement is certainly a lesson that different machine learning and high performance programs can benefit from. Given more time, it would be interesting to analyze the side effects of batch composition in a more granular scale, and perform an in depth computational analysis that could bound the number of Reduce operations. This would be tedious work! It was sufficient in the mean time to analyze the empirical results.

Acknowledgements

I would like to thank Samuel Bowman who was kind and informative when discussing SPINN, and James Bradbury who offered helpful advice on implementing SPINN in Chainer and suggested the peano function when bucketing sentence pairs.

References

[Bow+15] Samuel R. Bowman et al. "A large annotated corpus for learning natural language inference". In: *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, 2015.

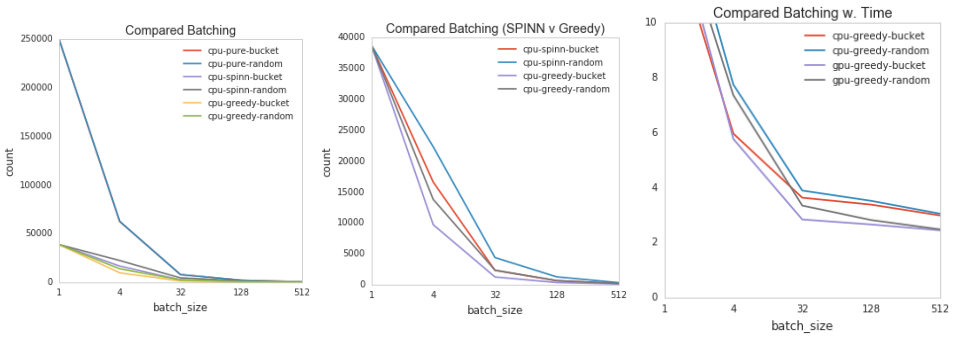

Figure 3: These plots are used to demonstrate the impact of different batching schemes (bucketing or random). The left two plots show the number of reduces on only CPU models (as the number of reduces will not change between CPU and GPU). The plot on the right is the total time to complete using the different batching schemes.

[Bow+16] Samuel R Bowman et al. "A fast unified model for parsing and sentence understanding". In: *arXiv preprint arXiv:1603.06021* (2016).

[Yog+16] Dani Yogatama et al. "Learning to Compose Words into Sentences with Reinforcement Learning". In: *arXiv preprint arXiv:1611.09100* (2016).