On Assaying the Parallelism of Neural Network Algorithms

Final Report
EECE 5640, Prof. Leeser

Samuel Sussman, Oleg Vaskevich

April 29, 2015

Project description

Our project dealt with improving the performance of Google's `cuda-convnet2` convolutional neural networks (CNN) framework by applying the various types of parallelism that we have learned in this class. `cuda-convnet2` is written in C++/CUDA, and is decently complicated, containing tens of thousands of lines of code. Its implementation as well as the mechanisms of its contributions are thoroughly described in [1].

In order to measure the performance of `cuda-convnet2`, we used a subset of the *LSVRC-2012* dataset, available from *ImageNet*. At a high level, there are two main components within `cuda-convnet2`: the data generation code, and the neural network training code. Initially, we looked into parallelizing the latter; indeed, we got in touch with the original author, Dr. Alex Krizhevsky of Google, in order to see where we can make improvements. However, we noted that since this part of the framework was already written in CUDA, our exposure to parallelization mechanisms would essentially be limited to just MPI.

As a result, we looked into the data generation portion of the framework. This sequential code was written in Python, although it did include a library written in C++ that used native threads for some basic parallelization. This dearth of parallelization provided for a great segue into the motives behind this project, and allowed us to explore MPI, OpenMP, and CUDA simultaneously. Furthermore, focusing on this part of the framework allowed us to aim our efforts at high-performance computing and not at understanding the inner machinations of CNNs.

In particular, we explored methods of speeding up the processing the initial raw dataset—a brobdingnagian 140 GB tar image—into batches of JPEG images that can be parsed by the CNN trainer. We thus set on our path by implementing MPI for parallelizing the Python data generation code, and further improved the code's performance by rewriting some parts using OpenMP as well as using CUDA. Although the data-generation code was I/O heavy—having to process copious amounts of data—our results were promising and we were able to observe general performance improvements as a result of our modifications.

Sequential code

The original sequential code is available online[1], licensed by Google under Apache License 2.0. We started with this code and modified it in order to run on the Discovery cluster, which was complicated due to the various library requirements and prerequisites. In particular, we had to address the following issues:

- We had to load the appropriate modules (`python`, `cuda`, `atlas`, `opencv`) as well as set the appropriate include and linker paths;
- The Python 2.7 module was built using position-dependent code that isn't compatible with this library. We resorted to using the system-available Python 2.6 library that was correctly compiled with `-fPIC`;
- The version of `libjpeg` available on the system, *v6*, was incompatible with the library, resulting in compilation errors. We added *v8* of the library source into our repo; and
- There were issues with building because the correct OpenCV headers weren?t being included; we modified the appropriate files.

Working with Dr. Nilay Roy, we also had to install the Pillow Python module on the Discovery Cluster, as well as ascertain that we can use the `/scratch` folder for storing our data and other intermediate files.

Since we had decided to focus on the data generation portion of the code, it was helpful to know what this sequential code was doing. We found out that essentially, the Python code goes through every sub-tarfile in the original dataset, locates each of the images, shuffles them into random batches, resizes the images using a native C++ library (using OpenCV libraries for the image operations and *pthreads* internally for some parallelization), and then pickles—writes the binary representation of—the images to disk. Clearly, this code is still processing-intensive, and could be improved through parallelization.

In order to test the runtime of the code, we had to modify our data set to make it more feasible to test. This is because processing of the entire 140 GB dataset took several hours, which is not conducive to performance testing for our project, and also is profligate in regards to usage of the Discovery cluster. Luckily, since this tarfile itself contained a miscellany of smaller tarfiles, we were able to remove all but one, and were left with an 8 GB dataset, much easier to test. Coupled with the other 7 GB validation image dataset, we were down to processing about 15 GB of data. We will discuss the performance later, but processing this data using the original code took 257.4 s on the Discovery cluster.

Parallelism

Although we spent a good chunk of time analyzing the C++ code for training the neural network, parallelizing it required at least a thorough understanding of convolutional neural networks, and we haven?t gotten to the part where we understand them well enough to see how we could parallelize them. The code is already written in CUDA, and from our analysis of the CUDA code, there aren't really any improvements we can make with regards to optimizations such as shared code and tiling: the authors of the original code

[1] `https://code.google.com/p/cuda-convnet2`

seem to know what they're doing.

MPI

We started out by implementing the data-generation portion of the code with MPI. In order to generate the training batches from the raw data, the code takes a .tar file containing many .tar files of JPEG images of varying sizes, and extracts and resizes the images to 256×256 pixels. Since this process takes several hours on the sample data, we added support for MPI using *mpi4py* to speed up batch data generation. In particular, for our implementation we broke up the extraction onto different nodes in the cluster and rewrote the randomization and shuffling code to work around the issue of file handles being distributed across different nodes.

With respect to *programmability*, the *mpi4py* bindings were decently easy for us to use, and very similar to the C++ framework that we are all familiar with. However, we ran into a few snafus, and the most difficult part was synchronizing the status of each individual node of processing its individual set of images into a global percentage that the user can see; we ended up commenting this code out.

With regards to *portability*, the framework as a whole is not portable. This is because we had to perform a lot of changes in order to get `cuda-convnet2` to run on the Discovery cluster, and there may be changes required in order to run the framework on other systems. That being said, the data generation portion is decently portable. This is because *mpi4py* is readily available and easily installed.

With regards to *performance*, MPI provides a significant gain because it lets us use several nodes simultaneously for computation. By leveraging the resources of multiple machines on the Discovery cluster, we can significantly increase throughput.

OpenMP

Having completed the parallelization of the Python code with MPI, we next looked into what we could do about the C++ library that it loads to perform image processing resizing. The original code used an abstraction of pthreads—using 8 threads in particular—and we were interested in seeing how OpenMP would allow us to improve performance by letting us tweak the number easily, as well as make the code more maintainable. Of course, OpenMP is internally based on pthreads, anyway, but it did result in significantly decreasing code complexity.

With regards to *programmability*, OpenMP was fairly easy to use, since we really just had one for loop here. We did have to be careful notate critical code where needed.

With regards to *portability*, the same concerns as the ones for MPI are present here, but again, the ubiquity of OpenMP makes this code fairly portable.

With regards to *performance*, OpenMP lets us easily change the number of threads on each run of the program via environment variables, and we can empirically run the code to determine the best number of threads with which to run it.

CUDA

With MPI and OpenMP code behind us, we decided to look into using CUDA for the native C++ portion of the data-generation code. We noted that the most processing-intensive part of this code was decoding the image, resizing and cropping it, and writing it back to disk. That being said, this functionality was not explicitly implemented in the code, as it was using the OpenCV libraries. Luckily, the

OpenCV libraries provide a way of running `resize` with CUDA. We had to make sure to upload the decoded image onto the GPU, if there was one available on the system, conditionally call the GPU version of `resize`, and then download the resized image from the GPU.

Since the CUDA code was based off the OpenMP code, we were also able to run our CUDA code with a varying amount of OpenMP threads on the appropriate queue.

Testing

Instructions for running the code are available in the `README.md` file in the ZIP file or on our public GitHub repository.[2]. Note that initially we started by having four separate branches: one for the original code modified to run sequentially on the Discovery cluster; one for the the code with *mpi4py* implemented; and the remaining two for OpenMP and CUDA implementations on top of the former. However, we were able to combine these all into one branch.

This way, we can run the code with different behaviors as follows:

- To run the code sequentially, simply modify the Discovery Cluster Bash file to set the number of nodes *n* and *ptile* both to 1, set `NUM_WORKER_THREADS` in `make-data.py` to 8 (as was the default), and run on a non-GPU queue. Note that the original *pthreads* implementation was erased, but it should be trivial to merge the original code in and add a conditional flag to run the original version of needed.

- To run the code with MPI, set *n* and *ptile* as desired, modify `make-data.py` as described above, and run on a non-GPU queue.

- To run the code with OpenMP, run as above, but vary the value of `NUM_WORKER_THREADS` in `make-data.py` as desired. Run on a non-GPU queue.

- To run the code with CUDA, set `NUM_WORKER_THREADS` in `make-data.py` to 1 and run on a GPU queue like `par-gpu`. The code now automatically detects GPU availability and runs operations on the GPU. You can also modify `NUM_WORKER_THREADS` as desired.

In order to time the code, we used Python's native timing API. We simply measured the time at the beginning of program execution and at the end, and printed out the difference to standard output.

Furthermore, we wrote a test harness in Bash in order to run the code with varying numbers of nodes (`BSUB -n`), `ptile` and OpenMP threads, across multiple trials, as well as to extract this information into a text file. Since this tends to take a long time, we initially added a call to the Plivo SMS service in order for us to get text messages on our phones once the runs have completed. However, we couldn't use it because we found out that we had to run the Bash script on an interactive node, which doesn't have an internet connection.

Note that in practice, for unknown reasons some jobs ended up getting stuck and as a result, the Bash script didn't work too well with recording a lot of data. Jobs would also randomly fail, only to succeed when run again. We investigated, and this did not seem like an issue with our code. As a result, we ended up

[2] `https://github.com/ovaskevich/convnet-nu-discovery`

collecting less data for various combinations of n and *ptile*.

Results

Upon modifying the code to use a MPI, OpenMP, and/or CUDA, we ran the same data with varying combinations of processes, ptile values, number of threads, and GPU support. The raw results are available in the tables in Appendix A.

The first trial we tried was running the sequential code under one MPI process, using the original *pthreads* implementation instead of OpenMP. We observed a speedup of 0.89—in other words, the MPI implementation was a bit slower than the original implementation. This is most likely due to the overhead added by using MPI; at the same time, this effect is quickly mitigated by the benefits of using parallelism with MPI.

Indeed, in the best case scenario, with 32 processes spread out among 4 nodes, the parallel code achieved a speedup of about 4.4. Even in the worst parallel case—2 processes spread out among 2 nodes—we saw a speedup of about 1.12.

As soon as we added in OpenMP to the portion of the code that resized the images, we saw even more improvement. When the OpenMP code ran with more than 1 thread, it ran faster than MPI alone. The speedup provided by OpenMP peaked at around 10 threads, fluctuating only slightly. In the best case scenario—32 processes spread out among 4 nodes using 1024 OpenMP threads—we achieved speedup of around 6.9. A graph illustrating these observations is shown in Figure 1.

We also experimented with running our GPU/CUDA implementation of the C++ *make-data* module. With a single-processor

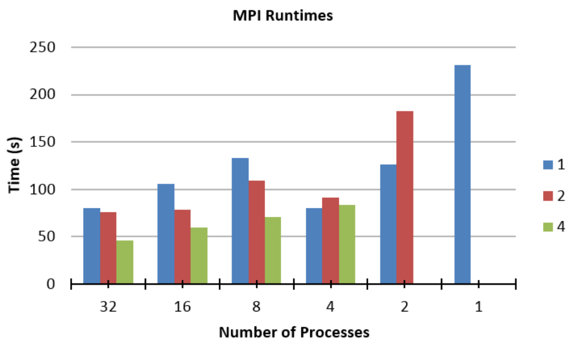

Figure 1: A graph comparing performance with MPI. The different colors represent the number of processors or nodes (number of processors divided by `ptile`).

execution, performance was similar to that observed with the same MPI setup. When we tried to use more than 1 OpenMP thread, though, performance quickly began to degrade. For instance, for $n = 16$ and $ptile = 16$, we recorded a time of 243.08 s with 32 OpenMP threads, and a time of 120.06 s with 16 OpenMP threads. This is likely due to the over head of uploading and downloading images to the GPU, when the actual resize operation is rather trivial. We found that running with with more processes resulted in better performance; unfortunately, the GPU queue was seldom not at full capacity, and we were limited in extracting measurements.

Conclusions

Our work for incorporating the various forms of parallelization that we learned in this class into a mature, sophisticated CNN framework proved most useful to augmenting our confidence in being able to apply high-performance computing techniques to even the most complex projects. By working with production code written in C++, CUDA and Python, we were able to learn how to apply the various frameworks in such a heterogeneous system, as well as learn how to use `mpi4py`, the Python bindings for MPI. If we had more time and expertise, we would look into parallelizing the neural network training portion of the framework. That being said, we were pleasantly surprised to see that our modifications, had a decent impact on the performance of the code.

Future work

Further work for improving the performance of `cuda-convnet2` on the Discovery Cluster lies in implementing MPI within the neural network code in order to take advantage of multiple GPUs across different nodes. This is due in part to the differences in the architecture used originally and the Discovery Cluster; the authors of the framework simply use one machine with many GPUs installed, whereas the nodes on the Discovery Cluster only have one GPU per node installed.

References

[1] Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks." (n.d.): n. pag. *Blackboard, EECE 5640.* University of Toronto. Web. 1 Feb. 2015.

A Appendix A

Processes	Processors	Average Time
32	4	46.45333333
	2	75.91333333
	1	80.385
16	4	59.5
	2	78.34
	1	105.44
8	4	71.15
	2	109.28
	1	133
4	4	83.55
	2	91.47
	1	80
2	2	182.79
	1	126.66
1	1	231.4

Table 1: Raw data for the MPI code. Average time was computed based on the average over 1 to 3 (depending on how many failed) trials.

Processes	Processors	OpenMP Threads	Average Time
32	4	1	78.3
		2	39.98
		4	34.78
		5	33.49
		8	31.46
		10	37.07
		16	31.92
		20	32.85
		30	32.23
		32	31.5
		64	36.52
		256	29.895
		1024	29.84
32	2	1	73.12
		2	55.1
		4	51.055
		5	47.88
		8	46.88
		10	49.39
		16	49.78
		20	49.86
		30	50.05
		32	51.545
		64	48.34
		256	49.35
		1024	49.83

Table 2: Raw data for the OpenMP code. Average time was computed based on the average over 1 to 3 (depending on how many failed) trials.

Processes	Processors	Time (s)
16	1	104.17
	2	184.14
	4	126.66
32	1	Out of memory
	2	160.15
	4	114.87

Table 3: Raw data for the CUDA code.