2017

Analyzing the Impact of Concurrency on Scaling Machine Learning Programs Using TensorFlow

Sheyn Denizov
Lehigh University

Analyzing the Impact of Concurrency on Scaling Machine Learning Programs Using TensorFlow

by

Sheyn Denizov

A Thesis

Presented to the Graduate and Research Committee

of Lehigh University

in Candidacy for the Degree of

Master of Science

in

Computer Science

Lehigh University

May 2017

Approved and recommended for acceptance as a thesis in partial fulfillment of the requirements for the degree of Master of Science.

_____

Date

_____

Thesis Advisor

_____

Chairperson of Department

ACKNOWLEDGEMENTS

I would like to thank Professor Michael Spear for his contribution to the project idea, his guidance, support, and patience through the entire process. He was invaluable in helping this thesis be completed in time and in the right way. I would like to thank the Lehigh University Computer Science department for their guidance and wisdom in helping me finish my Master's degree and this project in only one year.

I would also like to thank my parents for their support and patience while I did all of the work in the completion of this thesis and my Master of Science in general. Lastly, I would like to dedicate this thesis to my grandfather, with whom I was very close and who showed me the strength to do the things important to me and to never give up.

TABLE OF CONTENTS

ABSTRACT

In recent times, computer scientists and technology companies have quickly begun to realize that machine learning and creating computer software that is capable of reasoning for itself (at least in theory). What was once only considered science fiction lore is now becoming a reality in front of our very eyes. With this type of computational capability at our disposal, we are left with the question of how best to use it and where to start in creating models that can help us best utilize it.

TensorFlow is an open source software library used in machine learning developed and released by Google. It was created by the company in order to help them meet their expanding needs to train systems that can build and detect neural networks for pattern recognition that could be used in their services. It was first released by the Google Brain Team in November 2015 and, at the time of the preparation of this research, the project is still being heavily developed by programmers and researchers both inside of Google and around the world. Thus, it is very possible that some future releases of the software package could remove and/or replace some current capabilities. The point of this thesis is to examine how machine learning programs written with TensorFlow that do not scale well (such as large-scale neural networks) can be made more scalable by using concurrency and distribution of computation among threads.

To do this, we will be using lock elision using conditional variables and locking mechanisms (such as semaphores) to allow for smooth distribution of resources to be used by the architecture. We present the trial runs and results of the added implementations and where the results fell short of optimistic expectation. Although

TensorFlow is still a work in progress, we will also address where this framework was insufficient in addressing the needs of programmers attempting to write scalable code and whether this type of implementation is sustainable.

Chapter 1

INTRODUCTION

1.1 Rationale And Significance of Machine Learning

In the past several years, the hype over Machine Learning and the development of artificially intelligent software applications has quickly permeated markets throughout the world. A major impetus in developing this type of technology has been the consumer always wanting more from the computer with which they are working. Touch, speech, and face recognition are just three of the basic capabilities that users want their machines to have.

In order to try to meet this demand, companies have begun heavy research and development into machine learning platforms and the creation of artificial intelligence initiatives. These initiatives have even become part of mainstream pop culture. For example, the unveiling of IBM Watson at a Jeopardy match back in 2011 is almost always the first thing that comes to people's minds when they hear the words "artificial intelligence". In addition, if you were to go to the social media profile of a company, one would always find links relating to how their company is helping the cause of furthering artificial intelligence. For some companies, this can simply be clickbait for visitors of the site, for others, it could be the expression of years of research and development.

In keeping with their reputation as one of the (if not the) biggest technology companies in the world, Google has been at the helm of this trend for many years.

Starting in 2011, they began developing their own private machine learning library called DistBelief. DistBelief was based on deep learning neural networks for creating smarter additions to their applications. The people in charge of this project were a group of computer scientists and researchers at Google called the Google Brain team. Google Brain is the all-encompassing name for Google's artificial intelligence project. This diverse group of Google researchers found that they could produce much better applications if they allowed the projects to be self-learning. Thus, Google ran with this and the artificial intelligence project that they build to help implement "smarter" versions of their diverse range of services.

TensorFlow has three main implementation libraries: one for Python, one for Java, and one for C. The Python library is the most extensive TensorFlow package currently available, which makes sense due to Python being the quickest and most portable language for machine learning research. TensorFlow already has some built-in threading capabilities that allow large-scale machine learning algorithms to be run. However, the libraries are not well developed and can be buggy at times. Thus, the reason for our research experiment is now clear.

1.2. Rationale and Significance of Concurrency

The reason why concurrency is so big is simple: we need to use all computational resources we have in making our programs run more quickly. In order to do this, we need to have multiple instances running at the same time (or around the same time) to utilize our growing computational power. For instance, if we have a 4-core CPU, we would want to utilize all the cores to get the most out of the computer's abilities and effectively

run our programs. This same thought process is given to the rationale of having multiple instances of a program running at the same time.

Concurrency is simply the idea that we can have multiple instances of the same program running and still achieve cohesion between them. The hard part is obviously enforcing the cohesion of the processes and ensuring that their results fit together and each thread does not harm another. However, very sophisticated working methods on doing this were thought out before machine learning was even practical. Thus, we can use these ideas that have already been in place for so long in improving our machine learning and artificial intelligence capabilities.

Nonetheless, concurrency research is still a very popular field of study and ways to improve our current methods have still emerged in recent years. Therefore, we can still try to improve on those older models and see how they can be fit to new fields of study. From this, we can also adapt further methodologies to handle the problems that we encounter. The basis for all of this, however, remains in the foundational theory of concurrency and all the good that it has done computing for the past few decades.

Chapter 2

BACKGROUND

2.1 Why Do We Need TensorFlow?

2.1.1. A Brief Introduction

As explained in Chapter 1, TensorFlow is an open source AI framework owned and operated by Google. As one of the current big thing in the AI community, it continually receives improvements by developers from all around the world who are all dedicated in helping advance this software package and AI in general. The main TensorFlow codebase is hosted in a Github repository at https://www.github.com/tensorflow/tensorflow. As of February 2017, there have been over 16,200 commits (changes) and around 790 contributors to the codebase, with the profiles of the people ranging from college students to experienced industry research scientists.

This is indeed a positive thing about TensorFlow as a machine learning tool: due to it constantly being open to change, improvement of the codebase is perpetual. However, just as with any open source software comes the risk of a change massively corrupting the codebase and causing problems. This is why the developers at Google tasked with leading the effort are also in charge of tracking changes and making sure that rollbacks in the code occur if they are needed. To help them in this monstrous task, they use the

GitHub issues feature to read any bugs or change requests that programmers or users of the software have submitted.

```
>>> import tensorflow as tf
>>> hello = tf.constant('Hello, TensorFlow!')
>>> sess = tf.Session()
>>> sess.run(hello)
'Hello, TensorFlow!'
>>> a = tf.constant(10)
>>> b = tf.constant(32)
>>> sess.run(a+b)
42
>>>
```

Figure 1: Sample Python code utilizing TensorFlow

Google has also dedicated many resources (including an official site) to explain to interested users how TensorFlow works. There is an official site for TensorFlow (www.tensorflow.org) that explains how to get and install it in addition to Udacity courses on how to use this extensive framework. Figure 1 above shows some sample Python code (as Python is the main implementation library for TensorFlow, after all) that uses a TensorFlow "session" to add print "hello" and add two numbers together. More about this will be discussed in the next section.

2.1.2. How TensorFlow Works

As Figure 1 shows, TensorFlow is a software library that can be imported when you need it. However, you must first install the library onto your computer using the relevant installation instructions found on the official site (www.tensorflow.org). It is available for

download on Linux, Mac OS X, Windows, and, since very recently, the Android mobile OS. Now let us explain the way the basic code shown in Figure 1 above works.

The code above shows the Python console open with TensorFlow imported. Next, to use this imported library, we must create an object, here named tf, which will be able to run the built-in commands that TensorFlow has. The most common way to start off any TensorFlow program is to create a session, called "sess" in the code above. A session is a class that runs TensorFlow operations and manages the context of those currently running operations. It also manages resources and tracks the flow of the operations are being executed. For example, when we set the variables in Figure 1 above to be constants, we need the session object to run the + operation in order to add them. Thus, the session is undoubtedly the most interesting object in the TensorFlow library and will be very important to the implementation of our research experiment.

Moving on from the very simple code shown in the above figure, TensorFlow is at its most simplified form a library that uses data flow graphs to perform the numerical computations given to it. These graphs have nodes (or vertices) that represent mathematical operations, while the edges represent the tensors, which are actually multidimensional data arrays, that flow between these operations. This view is important because it emphasizes the reusability of code due to the fact that the tensors can simply go to new operations (nodes in the graph) by traversing said graph.

This type of outlook on numerical computation is very helpful in visualizing a hot new trend in machine learning: recursive neural networks (RNNs). These deep learning structures are used in research applications ranging from spelling corrections to mining of data for search engines. The way they work is by recursively applying the same set of

weighting values to the structure and producing some sort of structured prediction to the output value of some variable-sized inputs. This prediction output is typically produced by traversing through that structure using some sort of topological order. Naturally, these RNNs are typically represented by a tree structure, a decision tree to be more precise. Thus, it can be clearly seen that TensorFlow's representation of how to perform machine learning operations on inputs is analogous to the typical visualization of RNNs. This could also explain why many of TensorFlow's common use cases, such as image recognition and handwriting recognition, heavily utilize RNNs in computing accuracy scores for the training sets.

An understandable question that can come to one's mind right about now is why are we using TensorFlow. Why this particular library when our experiment may not translate well, or at all, into other libraries/domains? The best answer to this question is that TensorFlow is most likely the best that we can do. As the brainchild of one of the best research teams in the world, TensorFlow is undoubtedly one of the most powerful machine learning tools currently at our disposal. Will there soon be something much better to emerge that could replace it? Possibly, but we must make do with the knowledge and resources that we have at our disposal at this moment and improve them as much as we can. That is why we need to use TensorFlow in our research.

2.2 Why Do We Need Transactional Concurrency (Or Concurrency at All)?

2.2.1. A Look at What Concurrency Can Do

Modern computation has complex computational problems that need to be solved in a "reasonable" amount of time. The definition of reasonable in this context varies, but as

any algorithms course will tell you, we need this time complexity to be some function of the input size. However, with datasets and computational inputs so much larger today than they were before, we cannot simply rely on a single machine to perform even the simplest task on an uncountable input. Thus, computer scientists have come up with the concepts of threading and concurrency.

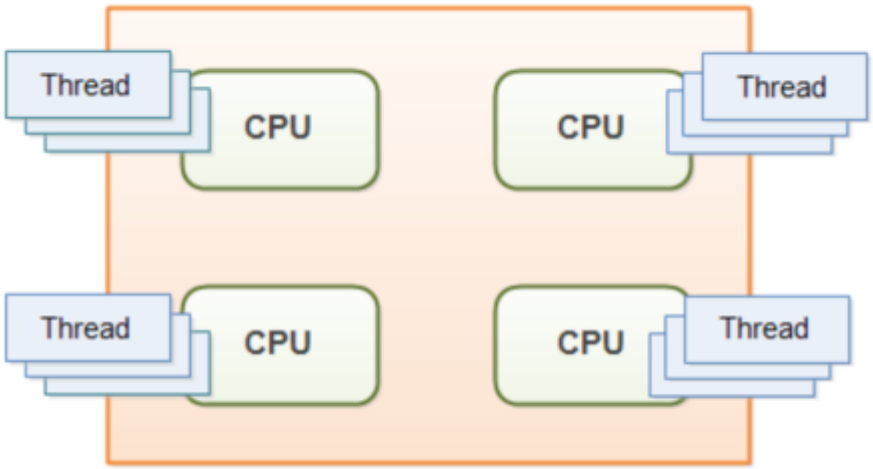

Figure 2: The concept of multithreading and CPU cores

The concept of threading is nothing new in computer science. For decades, computer scientists have known how multithreading and efficiently creating multiple instances of running applications could be a powerful tool in improving the runtime and execution flow of many algorithms. Figure 2 shows an abstraction of a computer, represented as the orange square, with 4 CPUs within it, which in this case could be thought of as cores of one big, "multi-core" CPU, and multiple threads given to each CPU. These threads represent running applications that each have their own context, their own low-level resources (or at least the illusion of them), and their own stack frames for execution.

Together, these parts of the multicore-threading model makes up the basis for the idea of serialization and concurrency of tasks.

However, we need to manage these threads and prevent them from conflicting when attempting to affect a shared resource item. The question now is how to do this. Could we allow limit the threads to a sequential order? No, because that would be an inefficient use of the CPU power that we have in being able to be run at the same time. Could we perform smooth context switches based on some interrupt? Maybe, but that could be expensive. Thus, we need some sort of way to make the wide array of concurrently running tasks and threads seem to run in an organized timeline. The answer to this is multithreading through conditional checks (through structures called conditional variables) and the use of locking structures to restrict resource access when they are being used.

The key to making sure that multithreading works well is to use it in the correct places. Typically, when running multiple threads of a program, we lock and unlock the data structures that we would like to amend when we are starting to use (i.e., amend or read) them and when we have finished with them (have finished reading and writing), respectively. We want to be sure that we are as close to certain of the value of a given variable at any point in time. In addition, we do not want to have values be corrupted by not being written completely. Thus, we would like a read/write locking structure to separate the two operations and allow permissions based on which one is being performed. For instance, if there is currently a write operation being performed, then we need a writer's lock to be acquired and to stop other threads from obtaining either a reader or writer lock.

2.2.2 Atomicity: The Transactional Concurrency Model

We described how the normal lock/unlock model works and how different conditions can prompt the acquisition of locks by different threads. For the most part, we can handle synchronizing the threads at a low level with little trouble, separating the two different operations, read and write, and harmoniously following the permission rules described above. However, how do we handle these locking structures when the operations and locks need to be acquired right away to prevent bad conflicts? A good solution would be to eliminate the low-level details by making everything done under locking "atomic". In this sense, "atomic" means all-or-nothing, where an operation is performed fully or it is not performed at all. This is called transactional memory concurrency.

Transactional memory is a concept borrowed from databases and database theory, and rightly so as the write operation needing the lock is like that of a transaction being performed on a database. To elaborate, a database transaction, that is, a change on some variable like the money in a bank account for a customer, needs to either completely be written to the database or not at all. If there were to happen something that corrupts or stops the transaction from being performed completely, then we will get rid of the part of the transaction that was written and remove it altogether. In addition, a customer looking at their bank account in the middle of it being written to will not see the transaction as it is being written. They will see an old version of their statement and will only see the changes when the complete write transaction has been performed. Thus, this all-or-nothing idea is what we would like to emulate in obtaining locks on shared resources.

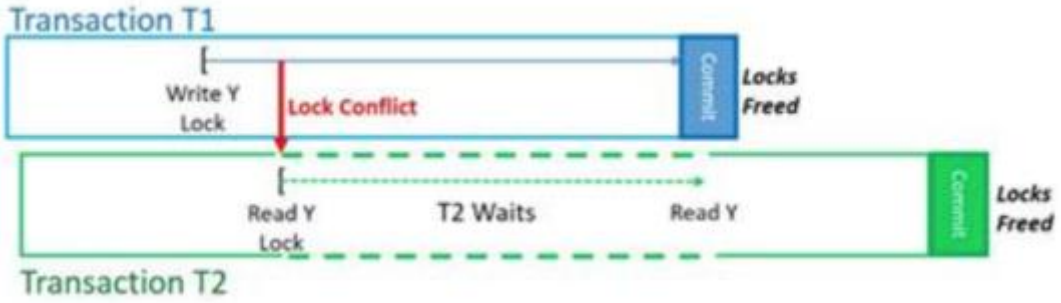

Figure 3: A model of the Transactional concurrency model

Figure 3 shows the transactional concurrency model. Two transactions, T1 and T2, are concurrently attempting to access the same shared resource. T1 has acquired a write lock to write to shared resource Y and later on in time, T2 would like to acquire a read lock for Y. However, T1 has not finished completely writing to Y and so T2 would be reading some corrupted value if it were able to open Y. Thus, a write lock takes precedence over read locks and a lock conflict occurs. During this time when T1 is still writing to Y, T2 waits for its opportunity to obtain the lock. In this case, it waits for the very first opportunity, which is when T1 commits its changes and signals that it has released its locks. When T1 finally does this, the signal is sent out and T2 can finally acquire a read lock on Y. Thus, as we can see in the figure above, a transaction must indeed be completed and committed completely before another thread can obtain any kind of lock (though multiple read locks can obviously be obtained when no one is writing). Otherwise, the transaction would abort and any attempted transaction operations would be rolled back (deleted). This model is therefore also good because it can relax deadlock due to the resolution of lock conflict being waiting for the first available opportunity to obtain a freed lock.

Transactional concurrency is an abstraction to the traditional model of locking and has some very good features. First, of course, is the atomicity of everything that happens. This factor definitely helps in our quest of almost always knowing the value of a variable or some shared resource when many threads are writing to it. In addition, because writing is a much costlier operation than reading, it will prevent us from having many corrupted writes and therefore save computational time and power, thus making the overall scheme more efficient. Furthermore, we save a good amount of hassle in having to worry about dynamically allocating locks to different threads as they emerge and managing how deeply their operations will affect their shared resource. This removal of burden can then allow the programmer to focus on making his learning model work and not on how to handle more threads producing surprising results at some points in time.

CHAPTER 3

THE METHODOLOGY

3.1 The Basic Algorithm

In order to implement this transactional memory idea into something as large as TensorFlow, we need to first understand how the basic idea will work. When setting down the idea, we need to be wary of how normal locking can be both beneficial and problematic. For instance, the main sources of lock contention can come from many threads attempting to do the same thing and simply overwriting each other. However, due to the speed of modern processors, we can be sure that basic TensorFlow applications will not be so heavily apparent. Nevertheless, there is a counterpoint to using transactional memory in designing this algorithm.

If we are attempting to speed up our TensorFlow session in which we have an image recognition tensor working on an input of many images, we will obviously use many threads. However, if these threads are constantly getting blocked by one another before they can even finish their operations, then the slowdown can pile up. Next, let us suppose we use locks, simple mutually exclusive locks. If we were to do this, we can see that our code has the potential to be much organized and synchronized. We would first allow a process to gain a lock, perform the critical section, and then unlock so another thread can go. However, it can soon become a hassle to manage all of these locking/unlocking steps and keeping track of which thread currently is reading or writing. Thus, there needs to be an addition to this that makes it easier.

Now we turn to our transactional concurrency idea from before. Unlike normal locking, we now use something called speculative locking. This is also known as lock elision. Lock elision is a type of mutually exclusive locking model that allows multiple threads to acquire the same lock, but prevents conflict by tracking what pieces of hardware are being modified (such as the cache lines of a program). This is something introduced in the Transactional Synchronization Extensions (TSX) architecture first introduced by Intel. This model will work well on supported machines as it allows hardware, and not just software, to come into the mix. The improvement of this design on most new hardware architectures is that it allows us to write code that does not have to worry about the individual lock working or not.

The basic algorithm for the lock elision that we used is shown below:

Program starts

Thread locks resource

Create an elided lock wrapper around the resource

If the transaction started

    If the lock is free

        Execute the lock

        Break

    Else

        Abort the transaction because lock is busy

        Delete partial writes

Perform the critical section

Go back to the initial setting of the lock and check for further transactions

```
Check for further lock requests
If the transaction is over
        If the lock is free
                Commit the transaction
        Else
                Unlock the lock and rollback any changes
Perform last operations that do not need locking
Program ends
```

The algorithm above makes use of a lock already existing on the resource and producing a speculative environment in which the currently waiting transaction can wait until a transaction is implemented. In truth, the algorithm is a wrapper class for the current lock or set of locks being held. The wrapper can then be reused so that whenever a lock request is triggered it can be called upon to check for whether that request can be readily satisfied. The biggest thing of note here is the abortion of the transaction if it has not been written entirely. Because machine learning algorithms such as those in TensorFlow require iteration and multiple concurrent writes over a very small period of time, we need a quick way to ensure that write locks are acquired at the right time and nothing surprising has happened to our shared resource. The use of lock elision will definitely help with that as it gives us more security in knowing that incoherent, unfinished writes have a smaller chance of being performed.

3.2 Fitting the Idea with TensorFlow

TensorFlow is a good candidate for using this type of concurrency structure. Since many workloads that can be created are not expressly built with scalability in mind, we can utilize the reusability of the concurrency algorithm to show whether we can scale these computational methods. Thus, we chose to run it on some workloads that were large enough to produce interesting results. These included image recognition deep learning tensors, handwriting recognition tensor sessions, and deep learning text analysis tensors. Naturally, there was also the need to create many threads that could work on these sessions concurrently to test the design. However, it is quite apparent that transactional concurrency using lock elision alone is not quite enough here.

In order to allow for architectures that do not necessarily have the full TSX extension in place, we need to ensure that the performance does not suffer as a result. Thus, we need to fall back on using conditional variables and semaphores to track the state of locks when they are done and report back when a waiting transaction can take place. In truth, we create a linked list of semaphores that can contain the state information of the currently processing transaction. For instance, if some thread A is currently writing to some shared resource, we implement semaphore S_A to be incremented or toggled as the transaction proceeds. Once the thread is done, semaphore S_A is put back in the linked list and marked as done. As we need more threads to run in a certain TensorFlow session, we will simply add more semaphores to the linked list and continue on in the same fashion. The same procedure can be used with conditional variables, although with the added benefit of using a broadcast unlock to unlock all the waiting threads when they are no longer needed.

Thus, we needed to implement our own small modified semaphore and conditional variable libraries to supplement the case when TSX was not in full use and complete lock elision could not be done properly. Nevertheless, the procedure of permissively allowing locks to be acquired without worrying about how and how many locks are obtained gives the programmer more freedom in writing the critical sections of their TensorFlow code. This can undoubtedly be good when we need to handle threads continually adding to some data structure and we simply cannot control these additions in a more feasible way.

CHAPTER 4

THE RESULTS

4.1 The Good and the Bad

To run the experiments, a machine with 4 cores and 16 GBs of RAM was used. In addition, TSX was compatible and we could see some positive change when attempting lock elision. Under these conditions, the speed of a TensorFlow experiment, without using transactional memory, was about less than 30 seconds for an image recognition experiment with n=10000 steps. This is a modest speed, given that certain machines built for this type of experimentation could do much better. However, this factor does not take away from the results and their foundations.

First, we start with the good things about our results. When a handwriting recognition algorithm was first run without any multithreading, it produced an accuracy score of 91.8% and took around 15 seconds to complete. When traditional threading (using simple mutex locks) with 5 threads was used, the accuracy score stayed about the same at around 92.5% and the runtime went down to about 10 seconds. Finally, when running that same program with the transactional memory model of lock elision and waiting locks, the time went down to about 5 seconds and produced an accuracy score of about 94.1%. When analyzing the throughput, which is the amount of time we spend on some factor per unit time, it seems to be almost linearly increasing as more threads are added with lock elision and going down as more threads are added with normal mutex locking. The table below shows some experimental results from training and testing runs of the multithreading

code and runs without it on a sample TensorFlow program that uses one session and a single method to recognize an image from the organization of the pixels.

Threads	Time spent without our concurrency model	Time spent with our concurrency model
1	30.15 seconds	31.20 seconds
2	34.57 seconds	30.50 seconds
3	36.31 seconds	30.08 seconds
4	37.25 seconds	29.55 seconds
5	36.47 seconds	29.21 seconds
6	37.54 seconds	29.07 seconds
7	38.45 seconds	28.17 seconds
8	38.51 seconds	27.54 seconds

Figure 4: Results of simple image recognition program running with n = 10000 iterations

As you can see in the table above, our model actually gets better time savings as more threads are added. This is in contrast to the time spent using the normal concurrency model, where it increases asymptotically after 2 threads. Though this speedup may not seem particularly large, it shows that the model is working correctly, that is, allowing lock acquisition to occur much more rapidly and with decreasing time overhead. Thus, the throughput, as mentioned above, is also much better, allowing more computations to take place in a much smaller amount of time. Even though there are times where the transactional lock elision model could and has encountered problems, such as deadlock if the critical section code is too convoluted or computationally intensive, for up to 8 threads it performs quite well on the image recognition example above, even at a high

number of iterations for training. Beyond this number of threads, as we are reaching the maximum number of threads of the machine we used, there can be some performance that makes the program run slower. This could be fixed with a more heavy-duty processor with more cores and is therefore not a problem in seeing that the model can scale if we had better hardware at our disposal.

A possible reason why this is the case is because mutex locking does not have a way to constantly check for resource availability. Because our lock elision has the added benefit of using semaphore structures to signal when a lock is free, it benefits from being able to not wait as long in the queue of processes waiting to run and also receiving priority. This is unlike normal locking mechanisms that employ the first-come, first-serve idea, or FIFO, to allowing locks to be granted upon a process. Hence, there appears to be some added benefit in using signaling and waiting data structures in our concurrency model, which is what we predicted when first deciding to use this methodology.

Next, we discuss the results that did not seem so promising. When testing the experiment on an experiment on large programs (i.e., TensorFlow programs using many inputs and thus many operations), the accuracy actually dropped in some cases. For instance, a deep learning image recognition algorithm that performed a large number of image recognition operations had its accuracy go down from around 80.2% to 76.5%. As was expected, lock elision encounters some problems when a great number of writes need to happen in very short periods of time. On the bright side, there were not a large number of deadlock situations in these larger programs. This means that the method will just need to be optimized and by no means scrapped.

4.2 Did Our Model Succeed?

The success of the research is dependent on two main things: whether the code could intermingle well with the design (proof of concept worked) and whether our initial thoughts are confirmed or disproven well enough. For the majority of test cases, the model worked and there appeared to be no real contention or deadlock to be had. The biggest cases where these problems would arise were, as implied above, when the RNN values were much too quick in their write transactions than the speed with which we could properly acquire locks and also when the program was much too big and extensive. These problems do serve the two functions above quite nicely. They show that the proof of concept did work in some cases and did not work so well in others. In addition, it proves our initial hypothesis that this model would work to some extent, but also disproves our notion that it would work for many more cases than it did.

Thus, it can be said that our model did succeed in its target aim of showing something new. It showed not only that transactional concurrency can work on certain aspects of this graph AI paradigm, but also that there are many more cases where it does not work and where we need to improve our current model of multithreading. The extent to which we can call this a success or is debatable. However, the experimentation was invaluable in showing how we could go about adapting the resources at our disposal to an ever-changing product like TensorFlow. Accuracy scores, precision, and succinctness of learning methods are all vital metrics in machine learning and they require multithreading in order to work well.

Luckily, TensorFlow does have built-in multithreading for the user to use. However, the more low-level we go with our analysis of the concurrency model that TensorFlow

uses, the more we can understand about how these threads actually run on these extensive tensor projects in the first place. From there, we can find ways to optimize them and better our knowledge of how multithreading works. Therein lies the point of this research and from that perspective, the model we created has succeeded in proving its stated point.

CHAPTER 5

CONCLUSIONS

The research into whether we can use transactional memory to speed up and improve the accuracy of machine learning processes has given both optimistic and pessimistic results. The transactional memory model is undoubtedly more efficient than traditional mutex locking in the context of what we used to test it. The inclusion of lock elision was helpful in showing how less direct control of the locks using the traditional mutex could allow for some gains in the throughput for running multiple threads. With little cases of uncontested deadlock, we can be more certain that this model does indeed bode well for the context of machine learning algorithms. In addition, with improvements to the model, we could definitely improve the performance of those programs under more threads. That, however, is work to be done in the future.

Further research will have to be done as to how well lock elision works on certain data structures and topologies. For example, TensorFlow uses graphs as its main data structure for visualization of how to apply its recursive neural networks. This structure is not ideal for locking and unlocking paradigms, as traversals of the graph are not sequential and can be unpredictable. This unpredictability will mean that writing reliable multithreading code will be more difficult. Since the TensorFlow model of machine learning seems to be growing in popularity, it appears that we will have to accommodate our multithreading models to it rather than vice versa.

Despite all of the work that remains, it can be conclusively said that concurrent machine learning is not an unsolvable problem. The results show that we can be optimistic about adapting our models to this field and our work can carry forward. It is therefore only a matter of time until this burgeoning field of study has gone beyond this model limitation and experiments such as this will be a trivial problem.

CHAPTER 6

FUTURE WORK AND CHALLENGES

6.1 Future Work

There is not currently much in the way of initiatives to bring concurrency to TensorFlow or machine learning. For many research scientists, the task of finding an efficient algorithm for deep learning and AI is in itself a wicked problem. As can be seen on TensorFlow's GitHub page, much of the concurrency code is currently being modified by public programmers who are trying to fix these problems simply as a result of their interest in the library's development. Thus, there is much still to be done in researching the effect of multithreading on the performance of machine learning algorithms.

Some good future experiments could be analyzing how we can tailor our current models to be expanded to this domain more efficiently. For example, the use of the transactional memory model in our research, even if it was slightly modified to fit the use case, is not exactly a innovative new view as to how to handle concurrency. Though it worked for proving results and being an interesting foray of somewhat conventional means being used in a different context, there can definitely be a more open-ended effort in trying to establish a refined concurrency model to tackle machine learning constructs like neural networks and highly recursive data structures. Strict transactional-based locking just does not seem like a good method in this sense and should be changed to fit this model more seamlessly.

In addition, TensorFlow is still being developed and its implementations are not all optimized. Much work remains in improving this library and making machine learning a more understandable topic (i.e., neural networks remain a black box to both researchers and programmers alike). If we do not have a stable machine learning platform, we cannot hope to have a stable model for concurrency for it as well. Thus, there needs to be more study in how we can optimize the algorithms themselves as well. Nevertheless, if TensorFlow can develop and improve, there is no doubt that a new model of concurrency will come about to better serve this need as well.

6.2 Pitfalls and Challenges

We describe the challenges discovered by using the TensorFlow framework to run the concurrency code which we used. The first big challenge was the size of TensorFlow itself. As a big build when attempting to install it from scratch, it can overwhelm the disk space of a smaller sized hard drive. Thus, a more space efficient option would be to import only the parts of the codebase necessary to create the desired program. However, this can be time-consuming and resistant to optimization due to the size of the codebase and dependencies. The second big challenge is the computation power needed to run a simple session. As a Python library, it is no surprise that TensorFlow would be quite computationally expensive. However, the default setting for the GPU/CPU usages are almost unstable. They allow almost all readily available processor power to be assigned to the currently running task. Thus, it is almost required that the user change the setting so that the sessions do not completely block other running processes (especially when running multiple tasks).

Related to the computational power needed, the speed of a heavy TensorFlow session can get slow, especially when using multiple RNNs. Even when using locking to prevent multiple requests from conflicting, the speed of computation can grow even slower if too much is done at once. For instance, an image recognition program set to run with a small load of parameters can take as much as 10 seconds more to finish execution when running multiple threads. This is likely due to the RNNs not being calibrated to a more conservative value by default.

Another common pitfall and problem with this design could be the use of transactional locking in the incorrect places. Because multiple sessions can run independently of one another and almost intertwine if certain sub-steps are needed in the training process, there is the problem of locks conflicting over a resource that's not even shared. If this is the case, deadlock can come about due to simple negligence and not checking that everything needed for the transaction to fully perform was not locked down. Indeed, this one has happened more than once in the process of doing the research for this thesis. Nonetheless, the pitfalls and tribulations encountered were necessary in providing the results that were so needed for this research and were invaluable in learning more about how the model could be better used in this project.

BIBLIOGRAPHY

Martín Abadi, Ashish Agarwal, et al. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. USENIX Association, 2015.

Jeffrey Dean. Large-Scale Deep Learning for Building Intelligent Systems. In *Proceedings of the Ninth ACM International Conference on Web Search and Data Mining*. ACM, New York, NY, USA. DOI: 10.1145/2835776.2835844, 2016.

Victor Pankratius and Ali-Reza Adl-Tabatabai. A Study of Transactional Memory vs. Locks in Practice. In *Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 43-52. ACM, New York, NY, USA. DOI: 10.1145/1989493.1989500, 2011.

Torvald Riegel, Patrick Marlier, et al. Optimizing Hybrid Transactional Memory: The Importance of Nonspeculative Operations. In *Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 53-64. ACM, New York, NY, USA. DOI: 10.1145/1989493.1989501

Chao Wang, Yujie Liu, and Michael Spear. Transaction-friendly Condition Variables. In *Proceedings of the Twenty-Sixth ACM Symposium on Parallelism in Algorithms and Architectures*, pages 198-207. ACM, New York, NY, USA. DOI: 10.1145/2612669.2612681

APPENDIX A

Source Code

The source code for this project will be available on a GitHub repository at https://www.github.com/SheynD/TensorflowThesis. There you will find the codebase, instructions on how to access certain parts of the code and where they are, links to certain resources used, and how to run a sample prototype experiment.

BIOGRAPHY

Sheyn Denizov was born on February 17, 1994 in the country of Bulgaria. He immigrated to the United States in October 2000 with his mother and father. He then attended Pen Argyl Area High School in Pen Argyl, PA and graduated in 2012. Although interested in computers from an early age, he did not start programming until his senior year of high school. He then went on to study Computer Science and Business at Lehigh University and graduated with a B.S. with Honors in the program in May 2016. He also studied Finance and earned a B.S. in that in May 2016 as well. Sheyn very much enjoys creating web applications and low-level programming using C and C++ in order to see how things really work. Apart from his great interest in technology, he enjoys soccer, hiking, and holds a black belt in Tae Kwon Do.