Parallelized NLP Market Prediction

E4750_2017Fall_PSMP_report
Samuel Beaulieu srb2208
Columbia University

Abstract

Many consider the stock market to be efficient, however, many of these efficiencies are based on numerical data (ie. past values and financials). There is, however, a wealth of textual data which influences investors' decisions and may lead to inefficiencies as it is hard to analyze quickly. This project explores how basic Natural Language Processing (NLP) algorithms might be parallelized to take advantage of these inefficiencies. It explores the challenges of memory management and limited string operations in parallelized NLP as well as its feasibility. Finally, we present a possible solution to these considerations and show that parallelization has the potential to speed up NLP by 1.5 to 10 times.

1. Overview

1.1 Problem in a Nutshell

Market efficiency, a theory developed by Eugene Fama in 1970, is based on the idea that all available knowledge about a stock is already included in its price [4]. It makes the argument that one cannot reliably beat the market because available data does not provide any indication of the stock's future movements. As shown in other publications, though, this theory is heavily disputed and has contradictory evidence [2].

We base this work on the assumption that, for capital markets to be efficient, information about a stock must be disseminated to the general population, consumed, and acted upon which requires time. Specifically, we take for granted that the market may be efficient in a long term sense, but not short term. The existence of algorithmic trading firms trading at high frequencies supports this theory as they are able to find and profit off of small inefficiencies. Though some take into account news articles and company information, many base their trades off numerical data such as past prices, trade volumes, technical analysis, and financial arbitrage [8].

Because of the algorithmic complexity and speed necessary to benefit from algorithmic trading based on numerical data, this paper instead looks at a different source of data: Text. We make the assumption that text data is much more challenging to process at high frequencies and that it is more implicative of how human traders will act. We base this assumption on the fact that humans generate and consume news articles, blog articles, company news, and social media and that they base their opinions about a stock's value on them. Such an assumption leads to the following theory: Textual data provides an untapped inefficiency in the market if it can be processed and acted upon quickly [5], [6]. Further, even if Natural Language Processing (NLP) is used to analyze text programmatically, if we can do so faster, those inefficiencies still exist.

Processing massive amounts of textual data, however, is not an easy task. Doing it fast makes the problem even more challenging [3]. In this project, we look at how parallelization using OpenCL and Python on modern General Purpose Graphics Processing Units (GPGPUs) can be used in Natural Language Processing (NLP) to make processing massive amounts of textual data feasible. GPGPUs are extremely well suited to processing huge arrays of numerical data which lends their use towards image and video processing, machine learning, and other similar applications. Unlike numerical data, text features variable length words which require different manipulation techniques. The main considerations we focus on are preprocessing, memory management, and string manipulation in parallelization kernels.

In order to analyze text, a considerable amount of preprocessing is necessary. Human text, especially in less structured cases like social media and blog posts, feature many abnormalities in terms of punctuation, use of numbers, and spacing. To keep things simple, we ignore these and strictly focus on the words themselves. Just splitting up the text into words, though, is a challenge in itself. Despite the time spent preprocessing the text, we found the gains from parallelization were enough to make a significant impact on the execution time of the algorithms.

Memory management is crucial when dealing with parallelization because each instance of the kernel needs to know exactly where in memory it needs to access without interacting with other kernel instances. Words and articles, by nature, have varying lengths which raises the question of how do we store the words so that they can be accessed uniformly by the kernel? The solution we use is to allot each word with a standardized amount of space in memory. By doing this, we're able to know exactly where each word begins.

The third major issue we tackle is string manipulation. In OpenCL, there is no way to import string libraries

which are typically used in string processing. As a result, there are no functions like strcmp(), memcmp(), strlen(), strcpy(), memcpy(), etc. Processing natural language requires comparing words and phrases quickly. Without these functions, that problem becomes more challenging. On top of not being able to use standard string manipulation functions, pointer arithmetic in OpenCL is extremely limited as well making it hard to refer to certain spaces in memory when dealing with words and phrases. Any parallelized NLP algorithms will have to tackle this problem before they are able to do anything else. In our case, we pulled individual letters from a character array and compared each one.

1.2 Prior Work

Using NLP to predict the stock market is not a novel idea. In 2001, Győző Gidófalvi from the University of California, San Diego, remarked on the exact assumptions examined in the previous section. He looks at the same market efficiency theory and sees a similar inefficiency in textual data being absorbed into stock prices, specifically financial articles [5]. In his work, he implements a naive Bayesian classifier to predict stock movements. His work is more focused on intraday news articles and prices as he ignores any articles from outside active trading hours [5]. The outcome of his work is positive in that he shows a correlation between the article and stock movement which provides validity to the assumptions we base this work on.

Similar work was done later in 2005 by researchers at The Chinese University of Hong Kong and The Hong Kong University of Science [6]. They similarly focus on the inefficiencies caused by human consumption of textual data and the delay in processing and acting upon it [6].

Based off these two works, is a final project, similar to this one, by Kari Lee and Ryan Timmons from the Stanford NLP department. Like the previous two papers, they use financial news articles to determine whether a stock will go up or down. They also implement a simulated trading platform based on the outcome of their algorithms which beat their benchmark of buying and holding throughout the trading process [7]. Unlike the two other works, theirs uses day granularity for stock prices due to limited access to intraday prices.

Each of these has many similarities to what we attempt to accomplish but they all focus on the actual outcome of the algorithms. In the case of this project, we focus on the potential to speed up these NLP algorithms to give a competitive advantage over human traders and other NLP prediction algorithms.

As explored by Chao-Yue Lai from the University of California at Berkeley, NLP algorithms are computationally complex and challenging to parallelize. His work focuses on parallelizing grammar analytics and language translation. Despite the complexity of the algorithms, he is still able to achieve a speedup of around 25 times [3].

2. Description

The following sections illustrate the objective of the paper, an overview of the flow of the project, the technical challenges faced, and the algorithms implemented. Specifically, in Section 2.1, we discuss what we are trying to achieve with a discussion on the different parts of the project, the parallelizable aspects, and the flow. In Section 2.2, we dig into the parallelizable portions and discuss the major challenges faced and how they were overcome. This leads to Section 2.3 which discusses the final software design and Section 2.4 which discusses the prediction algorithms in more depth.

2.1. Objectives and Project Design

The main objective of this project was to predict stock movements using parallelized NLP. Due to the lack of an open source, stock specific, timed, financial article database, we generated our own by scraping around 200 articles a day (10/stock for 20 stocks) from Google News and the links it provided. For financial price data, we scraped open and close prices from Google Finance.

Because of the limited database, we did not expect our prediction results to be accurate. Unlike the prior NLP stock market prediction efforts we explored in Section 1.2, we focus on the parallelization aspects of NLP. Stock market prediction is instead used to provide context for the project and a motivation for NLP to be sped up through parallelization.

Our focus on parallelization over algorithmic success means the algorithms we implemented are quite basic. Each relies on weighting each word of an article based on whether the stock the article was written about goes up or down. Every day, these words are updated with the results of that day and words that are not in the database are added. In this way, our model learns what words cause a stock to rise or fall and ideally can predict the direction a stock will take based on the words in the articles about it.

Of the seven prediction methods we implement, six are based on a very simple word weighting, while the seventh is a naive Bayesian classifier similar to Gidófalvi's work [5] explained in Section 1.2.

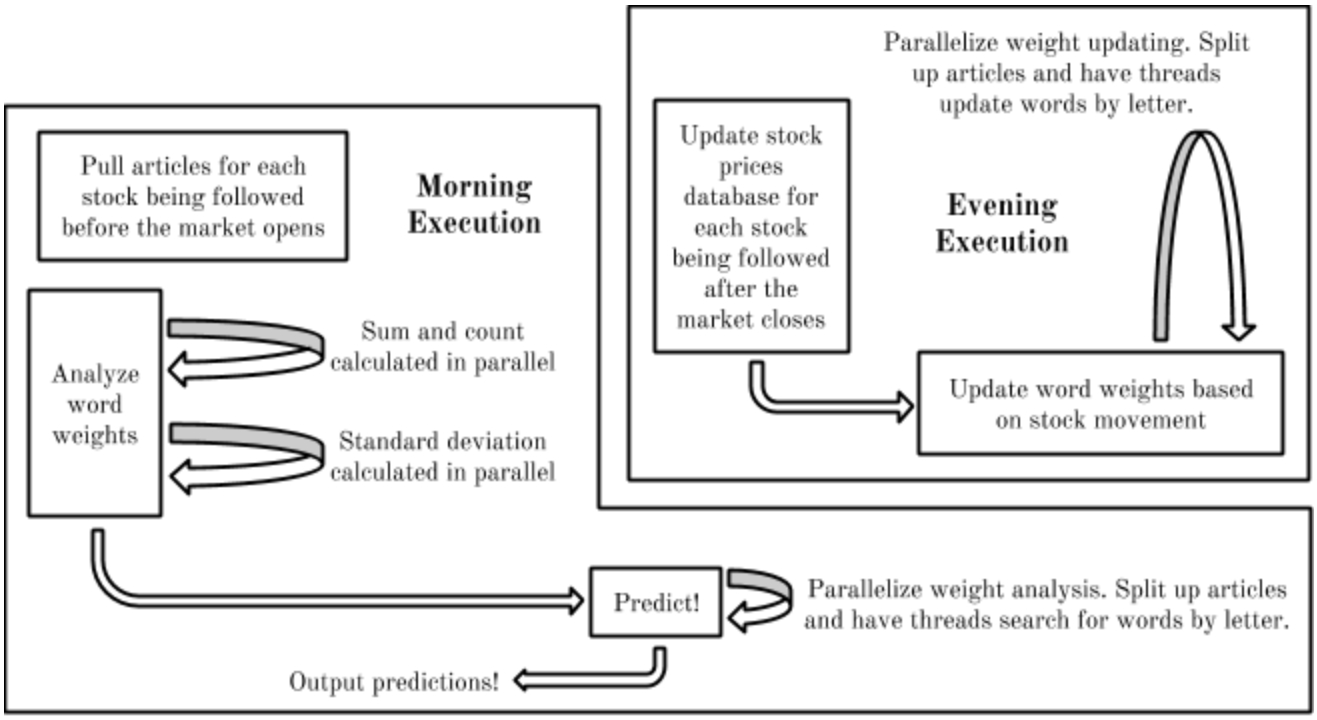

Figure 1: Block Diagram of Evening and Afternoon Program Execution

The execution of the algorithms can be split into a training phase and execution phase. In the training phase, word weights are updated based on the stock's actions the day after the article came out. Once the model is trained, the execution phase starts. In this phase, as shown in Figure 1, every morning before the market opens, articles are pulled for each stock and predictions are made. Then when the market closes, stock prices are pulled and the word weights are updated. As a result, the model continues to learn day after day and would ideally be able to adjust to changes in human behavior and societal shifts in writing.

There are three parallelizable parts to this design. They are: determining the weight statistics used in the first six algorithms, performing the actual prediction, and updating the word weights after the market closes. The key features that are required by these functions are to i) get all the weights associated with each of the words in the knowledgebase, ii) get each of the weights associated with an input list of words, iii) modify the weights associated with an input list of words, and iv) add words from an input list that are not currently in the knowledgebase to it. Each of these are examined in more detail in the following section.

2.2 Problem Formulation and Design

In this project, we use Python as our primary language for its portability and rich set of libraries. We use PyOpenCL for our parallelization code and chose it over PyCUDA so it could be portable to non-NVidia devices. The following subsections detail the different aspects of the project and any challenges that accompanied them.

2.2.1 Overview of non-parallelizable aspects

Using the blocks in Figure 1 as a baseline for the actions to be implemented, we first cover pulling the articles and stock prices. These functions are not parallelized and would ideally be provided through an external API. Both are written exclusively in Python and make use of many of Python's libraries.

To pull the articles, requests are sent out through Python's request library to Google News. We then use Beautiful Soup, another one of Python's libraries for HTML parsing, to parse through the returned HTML. We find links to relevant articles and then send out more requests to those links. Beautiful Soup is once again used to pull only the article text from those links and the articles are then saved in a Python dictionary by stock.

To pull stock prices, for each stock, we again make use of Python's requests library, this time making the request to Google Finance. Again, using Beautiful Soup we extract the open and close price of the stock and store

it in a dictionary associated with both stock ticker and date.

2.2.2 Overview of Parallelization Challenges

The three parallelizable functions analyze the weights, predict stock movements, and update the word weights based on the stock movements. The analysis function is fairly straightforward: each kernel obtains one weight from a weight array and performs reduction to get the statistics of the weights. The latter two functions rely on input words which have to be preprocessed into a format the kernels can use effectively. Once done, each kernel compares one input word with one word in the weight array. If they match, it returns or updates the weight.

This method of word comparison is not the most effective in terms of space, but it greatly saves time as each kernel does not have to search the entire word weight database. If each kernel got an input word and had to search the database, then more words could be processed but the kernel complexity would be higher and the time per word would be much greater.

What this means, however, is how the weight array and input words are managed in memory is extremely important as words can be different lengths and the number of total possible words is massive. Further, once each word is found in memory, they need to be compared without typical string comparison functions. These three issues: weight array management, preprocessing, and string comparison, and our solutions are explained in the following subsections.

2.2.3 Parallelization Challenges: Memory Management

Simply using one big character string, as words are typically stored, as an input fails due to each word being different lengths. Further, we wished to store the weights associated with each word alongside the word. Because Python typically uses memory differently from C, which OpenCL is based on, organizing the data so it could be accessed by both languages complicated the issue. Further, each input word needs to be compared with each word in the weight database, so some way to mitigate the total number of weighted words was also important.

Our final implementation of the weight database was to maintain 26 byte arrays of 70,000 bytes each. Each of the 26 arrays corresponds to one letter in the alphabet while every 28 bytes of the 70,000 corresponds to one word/weight combination. Storing data this way allows us to keep 2,500 words per letter and only make 2,500 comparisons per input word. In our testing, we found that this was a suitable maximum. We also maintained an array of 26 integers corresponding to the number of valid words in each byte array so invalid memory spaces were accessed by the kernels.

As shown in Figure 2, each 28 byte segment corresponds to 16 bytes for the word (which allows for 16 characters), a 32 bit float, and two 32 bit integers. Though some words might be longer than 16 bytes, standardizing the words at 16 bytes and truncating the ends allows us to get around one of the major issues with dealing with language on a GPGPU: dynamic word lengths. Further, standardizing the number of allowable words per letter makes indexing the array in memory much more manageable.

Figure 2: 28 Byte Segment in the Word Weight Array

2.2.4 Parallelization Challenges: Variable Word Lengths

Another major challenge was preprocessing the articles so each word in them could be compared to a word in the weight array. Similar to with the weight array, passing one large string to the kernel was infeasible due to varying word lengths. Kernels wouldn't know where words started or ended without iterating over the whole array or communicating with each other which takes away the advantage or parallelization.

As a result, we split up the articles into words and then allocate 16 bytes for each word in one large buffer. This way, each kernel can compare one word in each array as the start of each word is a standardized distance apart.

2.2.5 Parallelization Challenges: String Manipulation

With both the word weights and the article words able to be accessed by the kernel, the last roadblock was how to compare the words without using any functions from string libraries such as strcmp() or memcmp(). We eventually determined that the easiest, most effective way of doing this was to get each of the 16 characters in both of the word buffers and compare them for equality. Since any extra bytes are null characters, this technique works well.

2.3 Software Design

Each block in Figure 1 can be broken down more into a set of functions that implements it. These are explained more in depth in this section. As a general note, the code can be run on a standard computer or GPGPU based on the "GPU = True/False" flag at the top of the code. All the code, as well as a working collection of articles and output files can be found in this git repository: https://github.com/saribe0/elen_e4750_project

2.3.1 Pulling Financial Articles

The first block in Figure 1 pulls articles from Google News. Only one function is needed to implement this block. For each stock, it pulls the articles and writes them

to the file. The result is placed in a folder for the current day in the ./data/articles/ folder. These articles can then be used later to either predict stock movements or update word weights.

2.3.2 Parallelized Weight Analysis and Predictions

The second two blocks, analysis and prediction, are lumped together as one executable command whose flow can be seen in Figure 3:

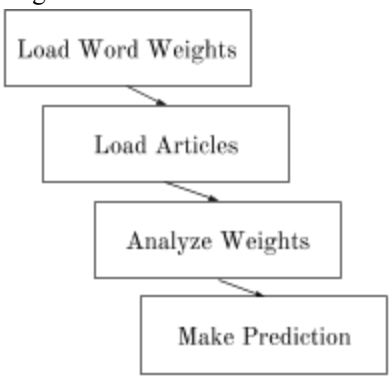

Figure 3: Analyze and Prediction Flow

The first step loads the weights from a text file generated after weights have been updated at least once. The second function loads the articles from text files for the requested day (more on exact commands in Appendix 8.1) and then the third function analyzes all the weights to determine things like maximum, minimum, standard deviation, and average. Finally, the prediction function gets the weights for all the words in the loaded articles, determines whether they're positive or negative based on the thresholds for that prediction method, and writes the prediction to a file in the ./output/ folder.

The analyze weights step includes two parallelized aspects when run on a GPGPU.

The first kernel uses 130 work groups of 512x1 work-items each. Each work-item gets one weight and stores it in local memory. Reduction is then performed and outputs in 6 arrays of 130 elements each which are then iterated through sequentially to find the overall maximum, minimum, count, and sum. Sum and count are determined for both unique words and all occurrences of the words.

The second kernel does essentially the same thing as the first, except it also takes both the average based on unique words and the average based on occurrences of words as an input. For each weight, it finds the square of the difference between the weight and each of the averages like so:

$$val = (weight - average)^2$$

$$val_{occurrences} = (weight - average_{occurrences})^2 * occurrences$$

The one using the average based on occurrences of the word is then multiplied by how many times that word has been seen during training. Reduction is then used to add up these products and the sums are used to find the standard deviation as shown in the following equation:

$$stdev = \sqrt{\frac{\sum_{i=1}^{N} (x - \bar{x})^2}{N-1}}$$

The pseudo-code for these two functions is shown in Figure 4. The flow of both kernels is similar. First, we find the indices necessary and get the weight and frequency from the weight array. Next, a local array is created and initialized with the starting values for that weight's index. In the first kernel, the six initializations are for the sum of weights, the sum of weighted weights, the maximum, the minimum, the count of weights and the weighted count. In the second kernel, the two initializations for the summation in standard deviation equation, one for the normal one and one for the one weighted by the words frequency. Then the reduction is done, synchronizing threads every step. Finally, the threads are synchronized one more time and then every work item writes back to the global output array. The output array is processed sequentially for the overall statistics.

The prediction step also includes two parallelized aspects. The first is for gathering the weights for prediction algorithms 1-6, while the second is for gathering the weights for the naive Bayesian classifier, prediction method 7. Both kernels do roughly the same thing except for how they use the data found at each word to generate the weight that is returned. For this reason, we won't focus on the differences until later on in this section. The prediction functions first preprocess the article data by iterating through the stocks and for each one, generate one big string of all the articles for that stock. It then separates them into individual words that are only at least 3 characters long and only letters using Python's regex library. Then all the words are saved into a byte array with each word getting 16 bytes as explained in the previous section.

The kernel is executed on a grid that is the number of words in the input articles rounded up to a factor of 256 by 2,560. 2,560 is similarly rounded up to the nearest factor of 256 from the number of possible words in the weight array per letter: 2,500. The group dimensions are not given as we let PyOpenCL determine them for us.

In the kernel, there are 2,560 work-items per input word. Each one attempts to compare its input word to one word in the weight array for its input word's first letter. The work-item that finds the matching word in the weight

Kernel 1:

```
__kernel analysis1(){
  // First, get the id of the weight and
  // the id of the letter of the weight in
  // the weight array. Also get the weight

  int w_id = global_id(0)
  int letter_id = global_id(1)

  if w_id < num_words_by_letter[letter_id]
     freq = weights[w_id + 1]
     weight = weights[w_id]/freq

  // Second, get the local indices, create
  // and initialize a local array
  // Size is work_group size by number of
  // statistics to be reduced

  int l_id = local_id(0)
  volatile __local float out[512*6]
  out[512*0+l_id] = weight
  out[512*1+l_id] = freq * weight
  out[512*2+l_id] = weight
  out[512*3+l_id] = weight
  out[512*4+l_id] = 1
  out[512*5+l_id] = freq

  // Perform reduction to combine all
  // values in the work group

  for s = 1 to 512, s *= 2

     // Synchronize the work items and
     // have the right items update
     barrier()
     if(l_id % s == 0 and l_id < 256)

        // Either add or find new max/min
        // arguments of max and min would
        // have 523*#+l_id as well if space
        // space permitted
        out[512*0+l_id]+=out[512*0+l_id+s]
        out[512*1+l_id]+=out[512*1+l_id+s]
        out[512*2+l_id]=max(out[+s],out[])
        out[512*3+l_id]=min(out[+s],out[])
        out[512*4+l_id]+=out[512*4+l_id+s]
        out[512*5+l_id]+=out[512*5+l_id+s]

   // Sync up the work items
   barrier()

   // After the reduction, write back
   if l_id < 6
     global[group_id*6+l_id]=out[512*l_id]
}
```

Kernel 2:

```
__kernel analysis1(){
  // First, get the id of the weight and
  // the id of the letter of the weight in
  // the weight array. Also get the weight

  int w_id = global_id(0)
  int letter_id = global_id(1)

  if w_id < num_words_by_letter[letter_id]
     freq = weights[w_id + 1]
     weight = weights[w_id]/freq

  // Second, get the local indices, create
  // and initialize a local array
  // Size is work_group size by number of
  // statistics to be reduced
  // w_avg is the average based on
  // occurrences

  int l_id = local_id(0)
  volatile __local float out[512*2]
  out[512*0+l_id] = (weight-avg)^2
  out[512*1+l_id] = (weight-w_avg)^2*freq

  // Perform reduction to combine all
  // values in the work group

  for s = 1 to 512, s *= 2

     // Synchronize the work items and
     // have the right items update
     barrier()
     if(l_id % s == 0 and l_id < 256)

        // Add up the values
        out[512*0+l_id]+=out[512*0+l_id+s]
        out[512*1+l_id]+=out[512*1+l_id+s]

  // Sync up the work items
  barrier()

   // After the reduction, write back
   if l_id < 2
     global[group_id*2+l_id]=out[512*l_id]
}
```

Figure 4: Pseudo Code of the Two Analysis Kernels

array returns the weight which can then be used by the prediction method.

The pseudo-code for the prediction kernels can be seen in Figure 5. Similar to with the analysis kernels, both of these are very similar. Both find the indices of the words and weights they are to compare, gets the words at those indices, and compares them. The difference happens when the weights are actually retrieved. The basic kernel

Kernel 1:

```
__kernel prediction_basic(){
  // First, get the id and index of the
  // word and the id of the weight

  int w_id = global_id(0)
  int w_idx = w_id*16
  int weight_id = global_id(1)

  // If the id is valid, get the word
  // and calculate the weight index

  if w_id < max_words
     int let_idx=words[w_id]-'a' or 'A'
     int weight_idx=
       (let_idx*size+weight_id)*28
     int weight_max =
       (let_idx*2500+num_w[lett_idx])*28

     char word_0 = words[w_idx+0]
       .
       .
       .
     char word_15 = words[w_idx+15]

     // If the index is valid get the
     // weight word

     if weight_idx < weight_max

       char weight_0 =
              weights[weight_idx+0]
              .
              .
              .
       char weight_15 =
              weights[weight_idx+15]

       // Compare words

       if word_0-15 equals weight_0-15

          // Get the weight and write to
          // output. Each is 4 bytes
          // while the chars are 1
          int freq=weights[20-24]
          float weight=weights[24-28]/freq

          out_weight[w_id] = weight
}
```

Kernel 2:

```
__kernel prediction_bayes(){
  // First, get the id and index of the
  // word and the id of the weight

  int w_id = global_id(0)
  int w_idx = w_id*16
  int weight_id = global_id(1)

  // If the id is valid, get the word
  // and calculate the weight index

  if w_id < max_words
     int let_idx=words[w_id]-'a' or 'A'
     int weight_idx=
       (let_idx*size+weight_id)*28
     int weight_max =
       (let_idx*2500+num_w[lett_idx])*28

     char word_0 = words[w_idx+0]
       .
       .
       .
     char word_15 = words[w_idx+15]

     // If the index is valid get the
     // weight word

     if weight_idx < weight_max

       char weight_0 =
              weights[weight_idx+0]
              .
              .
              .
       char weight_15 =
              weights[weight_idx+15]

       // Compare words

       if word_0-15 equals weight_0-15

          float up=prob(weights[20-24])
          float dwn=prob(weights[24-28])

          out_up[w_id] = up
          out_dwn[w_id] = dwn
}
```

Figure 5: Pseudo Code for the Two Prediction Kernels

stores the weight over frequency in the output array while the Bayesian kernel writes the probability of seeing that word given an up or down movement to the up and down output arrays. This probability is calculated using the total number of times a word has been seen when the stock goes up and the total number of words along with a smoothing factor. How we calculate the probability, why it's needed, and what it means are explained in the depth in the Section 2.4.

2.3.3 Pulling Stock Prices

The fourth block in Figure 1 is the update stock prices block which is executed after the market closes for the day. In this step, stock prices for the current day are scraped from Google Finance. This step contains three functions. The first loads the database of stock prices into local memory from a text file. The second pulls new prices for each of the stocks being followed, and finally, the third writes the stock price database back to the text file so it can be used another day or by some other command. As explained previously, this step has no parallelized aspects.

2.3.4 Parallelized Word Weight Updates

The fifth and final block in Figure 1 is the update word weights block. This is a crucial step as it is what allows the model to actually learn which words are associated with positive changes and which are associated with negative changes. The flow of this block can be seen in Figure 6:

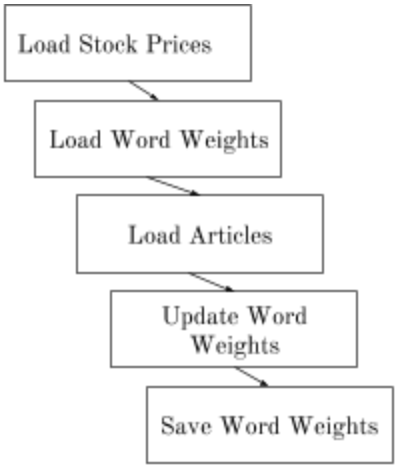

Figure 6: Update Word Weights Flow

The first three steps load all the necessary data to make the updates from their associated text files. This includes the stock prices to indicate stock movement, the previous word weights which are to be updated, and the input articles with the words to be updated. Once everything is loaded, the weights are updated and saved back to the word weight text file.

Aside from update word weights, the other four steps are all standard I/O. Update word weights is, therefore, the one we will focus on in this discussion. For each stock, it is determined whether the stock went up or down on the specified day and the articles for that stock on the specified day are assembled into a single array of words greater than 3 characters and without punctuation. Next each of the words is written to a 16 byte spot in a buffer. A list of integers is also created whose length is equal to the number of words found in the articles.

One of two kernels is then called depending on which weight type is to be updated (basic weights or Bayesian). Similar to the prediction kernels, the only difference between the two is how the variables inside the weight array are actually updated so for now, we'll focus on how the kernel works and not the actual numbers updated.

In the same way as the prediction kernel, the grid is defined and the work-groups are left up to PyOpenCL. Also similarly, each work-item compares its input word with one word from the weight array with the same first letter. A total of up to 2500 comparisons are made simultaneously and, when a match is found, the weight array is updated.

Since the weight array is shared by all the work-items, the updates have to be done atomically. This has the potential to reduce the advantages of parallelization, however, as explained in Section 3, we still saw a sizeable speedup from parallelization.

Aside from updating words that are found in the weight array, the update function must also add new words to the weight array. This is problematic to do in parallel for two reasons. The first is that if two work-items try to add a word with the same starting letter, they will not know that the other one is trying to do the same and will try and write the word to the same spot at the end of the weight array which causes memory conflicts. On top of this, if somehow this problem was avoided, then there is the issue of knowing that the word has not been found by some other work-item or already added by some other work-item.

Not being able to do this in parallel means the work-items need to indicate which words were found in the weight array and which ones were not. By doing so, the un-found ones can be sequentially added to the array. To do this, we initialize the previously mentioned array of integers to all zeros and use it as a bitmap. When a word is found by a work-item, the word's index in the bitmap is updated to 1. Since each word is in the weight array only once, only one work-item will write to any spot in the bitmap and memory access conflicts will be avoided.

The pseudo code for these kernels is shown in Figure 7. Both of these kernels are incredibly similar and, once again, only differ in what happens after the words are deemed the same. The rest of the kernels are the same as the prediction kernels so we won't cover that in depth again. If a match is found between the weight and the input word, the values in the weight array are updated. For the basic function, both the "up" and frequency counter are incremented for an upward stock movement and only the frequency counter is incremented for a downward stock movement. For the Bayesian kernel, the

Kernel 1:

```
__kernel update_basic(){
  // First, get the id and index of the
  // word and the id of the weight

  int w_id = global_id(0)
  int w_idx = w_id*16
  int weight_id = global_id(1)

  // If the id is valid, get the word
  // and calculate the weight index

  if w_id < max_words
     int let_idx=words[w_id]-'a' or 'A'
     int weight_idx=
       (let_idx*size+weight_id)*28
     int weight_max =
       (let_idx*2500+num_w[lett_idx])*28

     char word_0 = words[w_idx+0]
       .
       .
       .
     char word_15 = words[w_idx+15]

     // If the index is valid get the
     // weight word

     if weight_idx < weight_max

       char weight_0 =
             weights[weight_idx+0]
             .
             .
             .
      char weight_15 =
             weights[weight_idx+15]

      // Compare words and update
      // Each number is 4 bytes
      if word_0-15 equals weight_0-15

        if stock went up
          atomic_inc(weights+weight_idx+20)
          atomic_inc(weights+weight_idx+24)
        else
          atomic_inc(weights+weight_idx+20)

        bitmap[w_id] = 1
}
```

Kernel 2:

```
__kernel update_bayes(){
  // First, get the id and index of the
  // word and the id of the weight

  int w_id = global_id(0)
  int w_idx = w_id*16
  int weight_id = global_id(1)

  // If the id is valid, get the word
  // and calculate the weight index

  if w_id < max_words
     int let_idx=words[w_id]-'a' or 'A'
     int weight_idx=
       (let_idx*size+weight_id)*28
     int weight_max =
       (let_idx*2500+num_w[lett_idx])*28

     char word_0 = words[w_idx+0]
       .
       .
       .
     char word_15 = words[w_idx+15]

     // If the index is valid get the
     // weight word

     if weight_idx < weight_max

       char weight_0 =
             weights[weight_idx+0]
             .
             .
             .
       char weight_15 =
             weights[weight_idx+15]

       // Compare words and update
       // Each number is 4 bytes
      if word_0-15 equals weight_0-15

        if stock went up
          atomic_inc(weights+weight_idx+20)
        else
          atomic_inc(weights+weight_idx+24)

        bitmap[w_id] = 1
}
```

Figure 7: Pseudo Code for the Two Update Kernels

counters for up and down are incremented accordingly. After updating the counters in the weight array, the bitmap is set to one to indicate to the host the word was found.

After the kernel finishes executing, the program loops through the bitmap and uses the index of any elements set to 0 to pull the input word from the input word array and update it in the weight array. Unfortunately, this still has to loop through every word in the input array and potentially update them sequentially which limits the speedup that can be achieved.

In the worst case scenario that no words are found for a stock, each word will have to be sequentially added to the weight array. This will most likely only happen if the

weight array is uninitialized. Because the kernel is run once per stock, even if all the words are added sequentially for the first stock, it is highly unlikely that that is the case for the following stocks so a speedup, though small, is still possible.

2.4 Prediction Algorithms

As mentioned previously, we implemented seven prediction algorithms, six of which have weights based off of a basic weighting scheme while the seventh uses a naive Bayesian classifier. This section explains each weighting scheme and algorithm.

2.4.1 Predictions with Basic Weighting

The basic weighting scheme keeps track of two values: the total number of times a word is seen in an article of a stock that goes up, and the total number of times a word is seen in general. The weight is then the total number of times the word is seen and a stock goes up divided by the total number of times the word is seen. Each algorithm based on this weighting scheme uses some form of threshold based on an analysis of the weights and the average of the weights seen for that stock.

Further, some algorithms are not based on the raw average of weights themselves, but rather the number of standard deviations above the average weight. This was chosen to give some type of reference to how the market is generally trending and to provide some sort of indication of how much higher or lower a stock's rating is.

Each prediction method is indicated below:

Prediction 1:

This method uses weight analysis based on unique words when determining the statistics for all the weights. It classifies a stock as follows:

UP: If the stock's average word weight is 0.5 standard deviations above the all weight average.

DOWN: If the stock's average word weight is 0.5 standard deviations below the all weight average.

UNDETERMINED: Otherwise.

Prediction 2

This method uses weight analysis based on unique words when determining the statistics for all the weights. This algorithm only uses individual weights that are one standard deviation above or below the average weight when calculating the stock's average weight rating.

UP: If the stock's average word weight is 0.5 standard deviations above the all weight average.

DOWN: If the stock's average word weight is 0.5 standard deviations below the all weight average.

UNDETERMINED: Otherwise.

Prediction 3

This method uses weight analysis based on unique words when determining the statistics for all the weights. Weights are not influenced by how many times the word has been seen.

UP: If the stock's average word weight is above the all weight average.

DOWN: If the stock's average word weight is below the all weight average.

UNDETERMINED: Otherwise.

Prediction 4:

This method uses weight analysis that takes the frequency of the words into account when determining the statistics for all the weights.

UP: If the stock's average word weight is 0.5 standard deviations above the all weight average.

DOWN: If the stock's average word weight is 0.5 standard deviations below the all weight average.

UNDETERMINED: Otherwise.

Prediction 5

This method uses weight analysis that takes the frequency of the words into account when determining the statistics for all the individual weights. This algorithm only uses weights that are one standard deviation above or below the average weight when calculating the stock's average weight rating.

UP: If the stock's average word weight is 0.5 standard deviations above the all weight average.

DOWN: If the stock's average word weight is 0.5 standard deviations below the all weight average.

UNDETERMINED: Otherwise.

Prediction 6

This method uses weight analysis that takes the frequency of the words into account when determining the statistics for all the weights.

UP: If the stock's average word weight is above the all weight average.

DOWN: If the stock's average word weight is below the all weight average.

UNDETERMINED: Otherwise.

2.4.2 Predictions Using a Bayesian Classifier

The other weighting scheme is slightly different. Instead of keeping track of positive occurrences and total occurrences, it instead keeps track of positive and negative occurrences. This is then used to calculate the probability a stock goes up given the the word being seen and the probability a stock goes down given the word being seen. The highest probability is the one picked by the classifier.

Calculating the probability of seeing an up or down movement given the words for a stock can be expressed like so:

$$P(y|word1, ..., wordn) = \alpha * P(y) * P(word1|y) * ... * P(wordn|y)$$

This expression reads *"The probability of label y ("up" or "down") given words 1 through n is equal to a normalization factor, alpha, times the probability of label y times the probability of word1 given y times the probability of word2 given y, etc. all the way to the probability of word n given y."* Since the normalization factor is the same for both up and down probabilities, we can ignore it in our calculations.

Armed with this equation, all that is needed are the probabilities of the words and the probabilities of the labels "Up" and "Down." The latter are easy to determine. We simply keep a count of words in articles of stocks that go up and words in articles of stocks that go down and then divide them by the total number of words. This is shown by:

$$P(y) = \frac{[total\ words\ in\ articles\ of\ stocks\ with\ label\ y]}{[total\ words]}$$

The last piece to the puzzle is the conditional probability of the word given the label. This is, in fact, the "weight" returned by the Bayesian classifier kernel and can be calculated using the values stored with the word weights like so:

$$P(wordn|y) = \frac{count(wordn, y) + c}{[total\ words\ with\ label\ y]+c*[total\ words]}$$

Count of *wordn* with label *y* is one of the two values stored with the word in the weight array, total words with label *y* is, similar to before, the total number of words in articles of stock's that later went up or down, and total words is the total number of words. C in this case is a Laplacian smoothing factor to account for scenarios where the word has not been seen with the desired label but is still possible.

As one might be able to imagine, the probability of a word given an "Up" or "Down" label is probably quite low. When chained together with every other word in the article, a very small (un-normalized) probability is produced. So small, that it may not be feasible to compare the values of up and down. We fix this by transferring to log space and adding up the log of each of the probabilities which will instead equal the log of the conditional probability we are looking for. Since we don't care about the actual value, just the greatest, we keep the log of the value for our comparison and analysis.

As explained before, whichever probability is highest, ergo whichever log of the probability is highest, is the one the algorithm picks.

3. Results

In the discussion of our results, we first focus on the primary goal of realizing a speedup of NLP algorithms through parallelization and then transition to the secondary goal of predicting the stock market using NLP.

The GPGPU that was used for testing the parallelizable aspects of this project was a NVidia Tesla K40c which has 11,440 MB of global memory, a maximum clock rate of 876 MHz, and a memory clock rate of 3,004 MHz. It has 2,880 Cores which allows for a total of 2,048 parallel work-items and maximum of 1,048 work-items per work-group. The full details of this device are shown in Appendix 8.2.

3.1 Parallelization Results

3.1.1 Speedup From Parallelization

For the parallelization aspect, we look at all three kernel types and the difference between the speedup of the function realized by the kernel and the speedup of the entire function itself. The reason for this is to determine the effectiveness of the parallelized part of the code.

The timing we measure does not account for file I/O and the printing of results. Even though we recognize the extensive time these take (especially when run on the GPGPU), it is based strictly on how one chooses to obtain the data and process it, not whether the parallelization was effective. For instance, in a production environment the data might be piped directly to the functions and then the

outputs be directly used to take some kind of action thus removing the necessity for file I/O and print statements.

Execution Type	Avg. Kernel Speedup	Avg. Function Speedup
Analyze Weights:	82.487	1.604
Predict (Basic Weights)	36.354	6.402
Predict (Bayesian Weights)	30.462	4.647
Update (Min) (Basic Weights)	2.054	1.475
Update (Min) (Bayesian Weights)	2.047	1.470
Update (Max) (Basic Weights)	74.314	4.994
Update (Max) (Bayesian Weights)	73.794	4.941

Table 1: Speedups From Parallelization

Table 1 shows the average speedup achieved from running each of the prediction and update functions five times without the overhead of checking accuracy (commented out). With checking accuracy included, some of these values decrease, especially the value for the analysis function which becomes a bit less than 1.

Since each run consists of one kernel call for each stock and there are 20 stocks, each of the five runs equates to timing the GPGPU and CPU times 20 times on varying sized inputs. As a result, the numbers shown in Table 1 indicate the speedup averaged over 100 calls of the prediction and update kernels. Each of the analysis kernels was averaged over 10 calls.

It is also important to realize what is included in the function run time but not the kernel run time. Primarily, this is the preprocessing of the words and the analysis of what the GPGPU outputs as well as preparing the buffers for the GPGPU. For the functions with considerable preprocessing like the prediction and update functions, this includes creating one big array of valid words and writing all of these words to a single buffer in 16 byte segments. For all the functions, it means iterating through the output of the GPGPU and performing whatever calculations are necessary to determine the final output of the function.

Another important realization from testing is that varying input sizes affect how much speedup is achieved through parallelization. If the articles for a stock are on the longer side, the speedup will be greater as the sequential algorithm on the CPU will have more words to process one by one. The GPGPU on the other hand will not see an appreciable slowdown because of the massive number of cores and huge amount of memory available to it. Table 1 shows all the algorithms run on five typical input sizes of around 5,000 to 10,000 input words per stock, however, if the number happened to be more or less, the speedup could vary.

In a simplified example, consider the 11,440 MB of global memory on the GPGPU. If we take away the 26x70,000 byte arrays of weights that are passed to it and assume the smaller inputs negligible (an OK assumption as they are on the scale of 10s to 100s of bytes), the GPGPU is still able to hold over 300,000,000 words when running the prediction algorithms. This is far more words than we typically see in 10 articles. Of course, the actual number of words being processed at once would be far less due to core restrictions, but thousands of word comparisons would still be made in parallel compared to one at a time. This simple example is representative of the ability to scale up input size on the GPGPU with less overhead than on the CPU yielding increased gains.

Looking at Table 1, one can see that the range of the update function speedups is quite large despite being run on the same inputs. The discrepancy is a result of how the update function had to be implemented. Any words not found in the weight array at the time the kernel is called have to be added to the weight array sequentially. In the worst case scenario ("Min" in Table 1), the update function is run on an empty weight database. There is still some speedup because the kernel is run once per stock and there will be overlap in common words between stocks, but it is minimal. The best case scenario ("Max" in Table 1), is when every word in the input array is found in the weight array. As the table shows, the speedup is much better in this case. In reality, it is more likely for the speedup to be somewhere in between these two extremes depending on the amount of training the model has. A more mature model will exhibit greater speedups from the more complete weight database compared to a younger model.

3.1.2 Accuracy of Parallelized Parts

One consideration we had was whether we had a loss of accuracy as a result of the parallelization. Sometimes this can result from converting between Python's native data types and OpenCL's. For instance, depending on the system the code is run on, Python may use 64 bit floating

point numbers instead of the 32 bit floating point numbers that OpenCL uses. Throughout our many test runs, we were happy to see the percent difference between the CPU and GPGPU was negligible. Since the update function only updates integer values, we were able to achieve a 0% difference between the CPU and GPGPU values. The analysis and prediction functions, however, use floating point values for the weights, but the percent error was almost zero as well. In fact, for the analysis functions, the average percent difference was typically around one millionth of a percent while for the prediction functions, the percent difference was typically around one hundredth of a percent. One reason for the difference is that the prediction function passes more floating point values between the GPGPU and the host machine than the analysis function does which increases the total error.

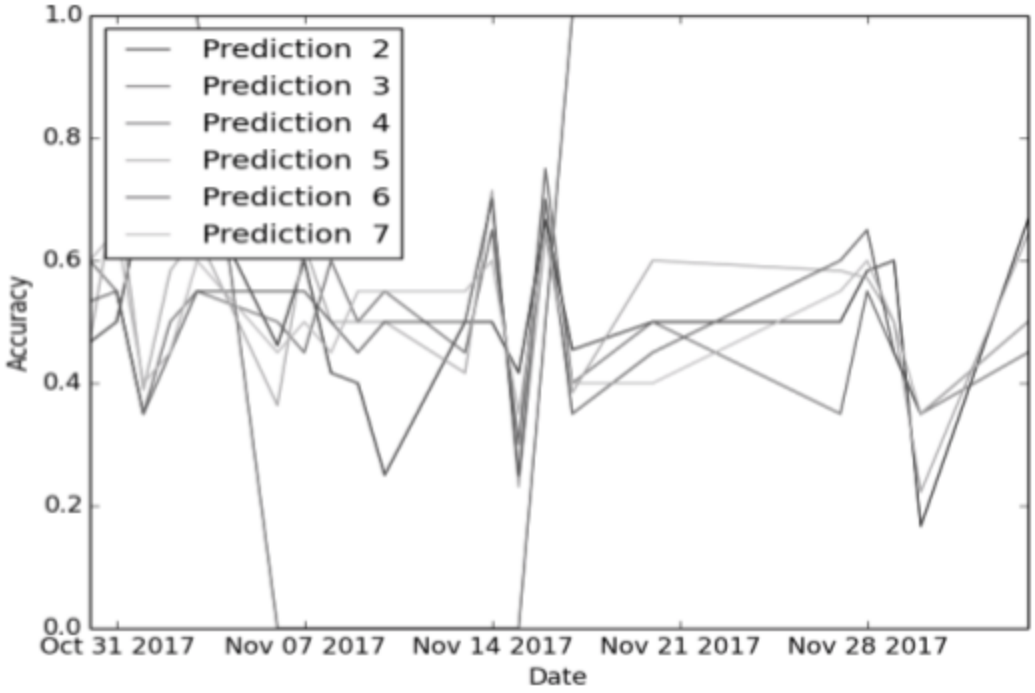

Figure 8: Prediction Accuracy From Late October Through Early December

3.2 Prediction Results

Figure 8 illustrates the success of the stock market prediction algorithms. The graph shows the percentage of accurate prediction for a variety of days from late October through early December. The initial training was for one day in late October and for each of the days after, a prediction was made and the word weights were later updated in the afternoon.

The graph clearly shows most of the prediction methods hovering around the 50% mark, however, there are some exceptions such as with Prediction 4. The graph only shows successful predictions which means "UNDETERMINED" predictions are not taken into account. If every prediction was "UNDETERMINED," it does not show up on the graph as with Prediction 1. Prediction 4 suffers from the same issue, except at some points it had one or two correct or incorrect predictions. In these cases, Prediction 4 either predicted a minimal number of stocks all correct or all wrong leading to 100% or 0% accuracy ratings on those days.

4. Discussion and Further Work

4.1 Discussion of Parallelized NLP

The parallelized NLP results we achieved seem reasonable and are of a similar magnitude to the ones achieved in previous parallelized NLP works [3]. Through them, one of the main issues of dealing with natural language in a parallelized manner is emphasized: Language and text is sequential by nature and is hard to parallelize.

Looking at the speedups achieved by the different functions, we can see that despite the huge speedup of ~82 times that the analysis kernel provided, the whole function was only sped up about 1.6 times which indicates the parallelized part was minimal and the pre and post

processing and preparation for calling the kernel greatly reduced the value of the parallelization.

Alternatively, by looking at the prediction functions, the kernel speedup was about 30 while the function speedup was around 5 which indicates that the prediction function benefited from parallelization much more than the analysis function did. Parallelizing the update function had an effect between these other two with a maximum kernel speedup around 70 and a functional speedup around 5.

As touched on in Section 3, the speedups for each of these functions seemed to improve when the input was slightly larger. This is typical of GPGPUs as they allow one to trade off area and parallelizable instances for time. Unfortunately, because of the way our function is designed the number of words processed is always roughly the same so we were unable to fully test how input size affects the speedup.

Going forward, it would be prudent to explore using larger input arrays of multiple stock's data. The biggest benefit to doing so would be fewer memory transfers between the host and GPGPU. These transfers are extremely expensive and cut down on the benefits of parallelization.

Limiting the pre and post processing could also give way to improvements. This could be done by keeping the input words in one big character array and using an index map of all the words. This way, different work-items would know where their word is and where the start of the next one is. Though more memory would be used on the device, as we showed before, we are currently not using much of it.

Though memory management was a huge challenge of ours, we can still see potential for improvements. Utilizing the faster, aggressively cached constant memory or somehow making use of shared memory for the weight array are places for further investigation.

Returning to the key points of our discussion of parallelized NLP from Section 2, we can now look at whether it actually makes sense to parallelize natural language processing. Though they are nowhere near what one might be able to achieve with massive amounts of numerical data, we were able to utilize the GPGPU to speedup a few basic algorithms a modest amount.

If we were to reduce the preprocessing through the methods described above (character array, more words at once, etc) or to obtain clearer articles and textual data from some sort of API or feed, the benefit of parallelizing the NLP algorithms could be increased.

Further, optimizing the kernels in more complex ways through memory usage and parallelizing more of what is currently done in post processing would reduce the data transfer between the device and host and cut down on sequential processing time.

Though the potential for parallelizing NLP is promising, we must also consider the drawback of the sequential functions we are comparing the parallelized ones to. If we were to rewrite the program in a faster, lower level language such as C or C++ and/or make better use of built in data structures like dictionaries or hash maps, the CPU runtime may also decrease. Pair this with a powerful multi core CPU of similar caliber to the GPGPU we are testing on and some of the benefits of parallelization may be mediated. We believe that we would still see improvements from the optimized parallelized implementation, but the magnitude may not be as much.

4.2 Discussion of Market Prediction Results

In terms of stock market prediction, the results are exactly what was expected. With minimal training data and very basic prediction algorithms, it would be shocking to see anything other than a random scattering of around a 50% success rate. If we continue pulling articles, and updating weights, it would be interesting to see if any or our prediction methods start to perform better than 50%.

Looking forward, there is potential to use text to predict the stock market. Past works have seen modest success and it is likely that large algorithmic trading firms have some form of textual analytics in place. Building off of this project, we can see a few places for improvement.

First would be to use more complex NLP algorithms that look into phrases, paragraphs, and nearby word associations. These would be able to provide more context behind words and hopefully indicate future stock movements better.

Second would be to reduce the granularity and look at intraday stock prices. Right now we make the assumption that the articles that come out before the market opens are not fully absorbed in the market's price until the end of the day and their influence will be effective regardless of the events of the day. Both of these assumptions have obvious flaws and experimenting with the prediction period of an article could show improvements in accuracy. This would also require using intraday article data and stock prices which would increase our database and improve our model.

If large amounts of data could be obtained along with intraday stock prices, an accurate model may be able to be trained and run throughout the day; quickly analyzing articles and social media and automatically responding to their release quickly and effectively.

5. Conclusion

Through this project, we set out to explore how parallelization could be used in the context of NLP. By focusing on the real life scenario of predicting the stock market, we provide reasoning for why NLP might need to be accelerated and are able to implement a few algorithms for testing. The main issues tackled by this paper are how to prepare textual data so that it is suitable for parallelization, how to manage memory so both the host and device can use it effectively, and how to deal with text data in the kernel, a task it is inherently ill suited for due to the lack of libraries and pointer control. We found that by standardizing memory allocations and using bitmaps, we are able to access data quickly within the kernel and pass information back to the host when needed. Also, even without string libraries, we were able to compare a collection of characters through simple base data types. Finally, despite the challenges of preprocessing the data, we found modest speedups were still achievable at a functional level.

Through the lens of algorithmic trading, implementing NLP algorithms in a parallelized manner could be extremely helpful. As technology improves and competition of the algorithmic trading sphere increases, speed is increasingly important [2]. If well thought out, we show that parallelization of NLP algorithms can be affective and could help traders tap into a wealth of textual data faster than their competition thus allowing them to profit.

When looking at the feasibility and effectiveness of parallelizing textual data analysis, it is apparent that, though there are benefits from parallelization, it may not always be the best choice. Text is, by nature, sequential. We speak and write word by word, letter by letter which makes it hard to parallelize. As a result, when deciding whether to parallelize an NLP algorithm, close attention has to be paid to how data will be organized in memory and which data structures are the most effective for both the host and the device. Developers will also have to consider how much of the algorithm is parallelizable and does the pre and post processing involved outweigh the parallelization.

6. Acknowledgements

I would like to thank both Professor Zoran Kostic and Professor Daniel Bauer from Columbia University. Professor Kostic supported this project and was our professor for the course that this project is for. Without him, this would not have been possible. Professor Bauer taught an Artificial Intelligence course that I took concurrently which influenced the algorithmic portions of this project. One of the assignments implemented a naive Bayesian classifier to classify short messages as spam or not spam in almost the same way as implemented here.

7. References

[1] Project Code:
https://github.com/saribe0/elen_e4750_project

[2] A. G. Titan, “The Efficient Market Hypothesis: Review of Specialized Literature and Empirical Research.” *Procedia Economics and Finance,* vol. 32, 2015, pp. 442-449.

[3] C. Lai, “Efficient Parallelization of Natural Language Applications using GPUs.” 2012.

[4] E. F. Fama, “Efficient Capital Markets: A Review of Theory and Empirical Work.” *The Journal of Finance*, vol. 25, no. 2, 1970, pp. 383-417.

[5] G. Gidófalvi, “Using News Articles to Predict Stock Price Movements.” 2001.

[6] G. P. C. Fung, J. X. Yu, and H. Lu, “The Predicting Power of Textual Information Financial Markets.” *IEEE Intelligent Informatics Bulletin,* vol. 5, no. 1, 2016.

[7] K. Lee and R. Timmons, “Predicting the Stock Market with News Articles.” 2007.

[4] M. J. McGowan, “The Rise of Computerized High Frequency Trading: Use and Controversy.” *Duke Law & Technology Review,* no. 16, 2010.

8. Appendices

8.1 Running Code

8.1.1 Directory Structure

The files and code included in the github repository is organized into three sections. First, there is the main code in the root of the directory along with the README and an accuracy graph (Figure 8) that was calculated from predictions made from late October through early December. The two folders are ./data/ and ./output/.

The ./data/ folder contains all the working data for the project. It contains one text file containing stock open and close prices for a variety of days. This file is updated when new stock prices are pulled and is called "stock_price_data.txt." It essentially serves as the stock price database's storage on disk. There are also two word weight database files: "word_weight_data_opt1.txt" and "word_weight_data_opt2.txt." These serve as the word weight database's storage on disk. Opt1 includes the word weights for the basic weighting scheme while opt2 includes the word weights for the Bayesian classifier. Each of these is loaded before a prediction is made and updated during the update phase. Also in the ./data/ folder is the ./data/articles/ folder. This folder contains one folder per day of downloaded articles, each of which contain one text file per stock labeled with the stock's ticker. These text files contain all the articles for that stock that were pulled on the day the folder is labeled with.

The ./output/ folder contains predictions made for all the prediction method and day combinations. The text files are labeled like so:

prediction<method number>-<date>.txt

The method number is two through seven inclusively with prediction one not having a number and skipping the field (a relic from before there were other prediction methods). The date is the date the prediction was made for and is formatted like so:

m-d-yyyy

The month is the one or two number abbreviation. For instance January => 1, *not* 01 and October => 10. Day is similar, with the fifth day as 5, *not* 05.

8.1.2 Main Commands and Options

There are many ways to run the code and they are covered in this section. For space considerations, we abbreviate ./stock_market_prediction.py to ./smp.py. When actually running, this would have to be expanded. Throughout these commands, weight options can be opt1 for basic weights and opt2 for Bayesian weights. All dates are of the format *m-d-yyyy.*

PULLING ARTICLES:

For the current day:
./smp.py -a

Supported Options:
none

PREDICTIONS:
(articles for the requested day must already be pulled)

For the current day with basic weights:
./smp.py -p

Supported Options:
-o <weight options>
-d <specified date>

PULLING STOCK PRICES:

For the current day:
./smp.py -s

Supported Options:
none

UPDATE WEIGHTS:
(articles and stock prices for the requested days must already be pulled)

For the update basic weights for the current day:
./smp.py -u

Supported Options:
-o <weight options>
-d <specified date>
-b <start date>
-e <end date>

If a start date is given without an end date, the end date will default to the current date. The specified date option does not work with the start or end date options. If there are dates within the date range (from -b date to -e date or current date) that do not have articles or stock data (such as weekends), they are skipped.

HELP:

To get this list printed:

./smp.py -h

WEIGHT ANALYSIS:
(must have weights calculated for the standard weight option)

Print the weight statistics:
./smp.py -z

DETERMINE ACCURACY:

Determine the accuracy of any predictions in the ./output/ folder:
./smp.py -v

This command also generates the graph of accuracy relative to the days the predictions were made.

8.2 Full GPGPU Data

Figure 9 details the full device information.

```
Device 0: "Tesla K40c"
  CUDA Driver Version / Runtime Version          8.0 / 7.5
  CUDA Capability Major/Minor version number:    3.5
  Total amount of global memory:                 11440 MBytes (11995578368 bytes)
  (15) Multiprocessors, (192) CUDA Cores/MP:     2880 CUDA Cores
  GPU Max Clock rate:                            876 MHz (0.88 GHz)
  Memory Clock rate:                             3004 Mhz
  Memory Bus Width:                              384-bit
  L2 Cache Size:                                 1572864 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536, 65536), 3D=(4096, 4096, 4096)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z):  (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z):  (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 2 copy engine(s)
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Enabled
  Device supports Unified Addressing (UVA):      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 4 / 0
  Compute Mode:
    < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >
```

Figure 9: GPGPU Device Information