Massively Parallel kNN using CUDA on Spam-Classification

Joshua M. Smithrud
Computer Science Department
Central Washington University
Ellensburg, WA, USA
Email: jsmithrud37@gmail.com

Patrick McElroy
Computer Science Department
Central Washington University
Ellensburg, WA, USA
Email: mcelrp@gmail.com

Răzvan Andonie
Computer Science Department
Central Washington University
Ellensburg, WA, USA
and
Electronics and Computers Department
Transilvania University
Braşov, Romania
andonie@cwu.edu

Abstract

Email Spam-classification is a fundamental, unseen element of everyday life. As email communication becomes more prolific, and email systems become more robust, it becomes increasingly necessary for Spam-classification systems to run accurately and efficiently while remaining all but invisible to the user. We propose a massively parallel implementation of Spam-classification using the k-Nearest Neighbors (kNN) algorithm on nVIDIA GPUs using CUDA. Being very simple and straightforward, the performance of the kNN search degrades dramatically for large data sets, since the task is computationally intensive. By utilizing the benefits of GPUs and CUDA, we seek to overcome that cost.

1 Introduction

A study estimated that over 70% of today's business emails are Spam (see (Blanzieri and Bryl 2008)). The purpose of email Spam is advertising, promotion, and spreading backdoors or malicious programs. The cost of sending Spam emails is much lower than the cost of automatically detecting and removing these emails. The processes of Spam-detection and filtration have become everyday occurrences within the average person's life. Though unbeknownst to most, each and every instance of digital communiqué goes through a series of automated filters to determine whether or not the user would have any interest in that particular item.

Because Spam-filtering systems are expected to be transparent to the user, there are two key aspects a good Spam-filter must contain: accuracy and efficiency. As the volume of email correspondences increases, so too does the amount of data required to properly classify emails into the categories "Spam" and "Ham" (non-Spam). The larger these training-sets become, the more time is consumed by the process. As a result, in order to prevent the process from becoming a burden to the user, Spam-filters must also be implemented more efficiently in order to compensate for the increased workload. No one wants to have to deal with Spam; the email client should handle it on its own. Our goal is to demonstrate the feasibility of an email client with hyper-accurate and hyper-efficient Spam-filtration capabilities, such that the standard email-user need not worry about the process.

A Spam-filter needs to sort incoming mail into wanted and unwanted, and it needs to do it accurately, which can be difficult. Machine Learning can help solve this problem: the email client can be trained to learn where to put each email. There are many Machine Learning techniques available to filter emails, including: Bayesian methods, Neural Networks, Support Vector Machines, and Deep Belief Networks, kNN, see: (Cormack 2008), (Tretyakov 2004), (Panigrahi 2012), (Blanzieri and Bryl 2008), (Sallab and Rashwan 2012). Actually, the most popular and well-developed approaches to anti-Spam are Machine Learning-based.

Filte *et al.* (Firte, Lemnaru, and Potolea 2010) proposed a kNN-based Spam-detection filter. In this scheme, messages are classified with the algorithm based on a set of features extracted from the properties and content of the emails. The training set is resampled to the most appropriate size and positive class distribution determined by several experiments. The system performs a constant update of the data set and the list of most frequent words that appear in Spam messages. The main reason for selecting the kNN algorithm was that it does not require a training step for each query. However, the method is offline and the authors focus on accuracy, rather than execution time. Actually, the kNN method can run quite slow, especially for larger values of k and when the input dataset is very large.

But how can we classify in real-time (online) incoming emails? Do we have to start the whole process from scratch each time? This is practically impossible when we have to deal with very large sets of emails. One solution would be to use incremental and parallel processing. Good scalability can be achieved by efficiently using parallel computer architectures like GPUs. Presently, GPUs are widely available and relatively inexpensive. In this context, the high-parallelism inherent to the GPU makes this device especially well-suited to address Machine Learning problems with prohibitively computationally intensive tasks. An increasing number of machine learning algorithms are implemented on GPU's (Lopes and Ribeiro 2011). At this moment, one of the universally accepted programming languages for GPUs is CUDA (Compute Unified Device Architecture) created by nVIDIA. Since its introduction in 2006, CUDA has been widely deployed through thousands of applications and published research papers, and supported by an installed base of over 500 million CUDA-enabled GPUs in notebooks, workstations, compute clusters and supercomputers[1].

Our goal was to design an email client with a highly-accurate, highly-efficient Spam-filtration capability. We propose a massively parallel implementation of Spam-classification using the kNN algorithm on nVIDIA GPUs using CUDA. The system we created can incrementally add new email samples to improve its accuracy. The proposed approach is not only fast but also scalable to large-scale instances. It also has the potential to scale for future devices with increasing number of compute units.

The rest of the paper is structured as follows. Section 2 describes how we extract features from emails. In Section 3, we introduce our parallel implementation of the kNN algorithm. Section 4 presents the architecture of the Spam filter, both from the designer and user perspectives. In Section 5 we describe experiments, discussing execution time and accuracy. Section 6 concludes with final remarks.

2 Feature Extraction from Email

An email message does not fall neatly into the domain of text classification. Email messages are much more than simple text, and must be handled accordingly. In addition to raw text, emails contain various elements of formatting (HTML tags being a notable example), meta-data, etc. In order to properly classify and filter emails, all of these elements must be taken into account.

In order for an incoming email to be classified as either Spam or Ham, it is first processed into quantifiable attributes. The attributes used are based on previous work in Spam Classification (Firte, Lemnaru, and Potolea 2010). Using MailSystem.NET's open source libraries, we extract the different components of the email message including:

- The number of recipients in the "To", "CC", and "BCC" fields
- The validity of the addresses in the "To" and "From" fields
- The validity of the "Message-ID" field
- The length of the "Subject" field
- The over use of capitalization
- The length of the message body
- The number of attachments
- The number of phony/suspicious HTML tags present in the email text
- The number of occurrences of the 50 most common Spam keywords in the message

These account for a total of 63 attributes. After being calculated, these attributes are normalized to weighted values between 0 and 1 based on the min and max values present for that attribute, such that no individual attribute contributes more to classification than another (this could be adjusted to fit specific environments if certain attributes are identified to be more useful than others).

The 50 keywords used for matching are determined based on the 50 most common terms found in emails classified as Spam within the user's account. In order to maintain an accurate list, the training set must be rebased with some regularity. Depending on the environment in which the application is being used, this can be done on an absolute schedule, or could be done as specified by the user.

The rebasing process accomplishes a couple things. First, the list of 50 keywords is re-evaluated. Using Lucene.NET's open source text analytics libraries[2] , we determine the 50 most common keywords found within the user's Spam folder. Once the new 50 keywords are found, the keyword counts for preexisting elements in the training set are recalculated. Second, training set values are normalized again (as new emails are added to the training set, it is possible for attribute values to exceed 1, so rebasing regularly becomes important to maintain consistency).

Basing the 50 keywords on the emails within the user's account allows for better personal accuracy. One man's Ham is another man's Spam (and vice-versa), so personalizing the training set is a valuable tool. There is a notable downside, however. Without a general consensus defining the difference between Ham and Spam, this system requires that the initial training set be adequately comprehensive, while being small enough that the user's preferences take precedence quickly. This makes the structure of the initial training-set used by new accounts critically important.

3 kNN Implemented in CUDA

Once an incoming email has been processed into its quantifiable attributes, it is used as the query point in our kNN classifier.

The kNN algorithm was implemented on almost all parallel architectures. A CUDA implementation of the kNN algorithm is described in (Arefin et al. 2012)). Under specific conditions, speed-ups of 50 to 60 times compared with CPU implementation were achieved, but this largely depends on the dataset and the GPU characteristics. There are also parallel kNN implementations which use the MapReduce paradigm (Yokoyama, Ishikawa, and Suzuki 2012). We will use here our own kNN CUDA implementation, specifically designed for the spam-classification problem.

For our work, we use sample emails from the SpamAssassin dataset[3] and stored them, after being processed, in an SQLite database[4]. Our implementation is divided into three phases: a distance calculation phase, a sorting phase, and a voting phase. Though the second phase is the primary focus of this work, we will examine each phase in detail.

The first phase is determining the distance between each point in the training set and the query point. Since this step is embarrassingly parallel, the process to determine the distances is simple and efficient. The query point and training set are transferred to the GPU, where the distance calculation kernel allocates a single thread for each data point in the training set. Each thread then simply calculates the Euclidean distance between its assigned point and the query point. The results of each calculation are stored in a struct

[1]https://developer.nvidia.com/

[2]http://lucenenet.apache.org/

[3]http://csmining.org/index.php/spam-assassin-datasets.html

[4]http://www.sqlite.org/

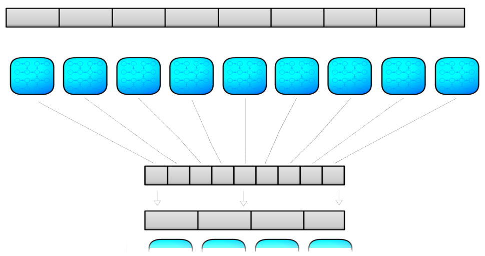


Figure 1: The input is divided among the executed blocks, with each block executing a reduction and returning the k smallest distances. Once all outputs have been collected, they are used as new inputs and the process repeats.

within the GPU's memory containing just the distance from the query point and the item's own classification. Since the attribute data itself is no longer of any use, we free it up such that the next phase of the algorithm can work on smaller data points. Additionally, the k data points we end up returning to the CPU are smaller and therefore reduced the overhead produced by transmitting over the PCIe Bus.

The next phase is the sorting phase. To improve the efficiency of the sorting process, we implemented a parallel reduction to determine the k smallest distances. Initially, the array of distances stored in the GPU's memory is used as input for the reduction. The array is divided among the blocks, and then the threads in each block perform a parallel reduction to find the smallest k distance within that block. Since we don't know where the smallest values are located among the blocks, and because threads cannot intercommunicate between blocks, each block must be reduced to its own minimal k. Once a block has determined its k elements of minimal distance, the results are stored in a secondary array on the GPU, of size equal to k * the number of blocks used. After all blocks have stored their results into the secondary array, the reduction is run again with the output of the previous reduction as the new input (juggling the reduced quantity of data points between the 2 previously allocated storage arrays (Fig. 1). This process is repeated until only k elements remain, at which point those elements are returned to the CPU.

The final phase is the voting phase. Since the number of data points used in the voting phase is k, which is generally set to a small value in order to maintain accuracy and avoid over-fitting, the computation for this step is negligible. For this reason, we execute this step on the CPU. The weighted distance for each of the k data points is added to its classification's total. Once each of the k data points has contributed its vote, the classification with majority would generally be determined to be the correct classification. However, since our implementation is designed specifically for use in Spam-Classification systems, a simple majority vote is not ideal. Generally speaking, it is preferable to have Spam emails incorrectly classified as Ham rather than the opposite (we prefer false-negatives over false-positives). As a result, our implementation utilizes a threshold value (θ), that the vote must exceed in order for an email to be classified as Spam.

4 System architecture

To demonstrate the functionality of the email classification scheme we developed, we created a lightweight email client, which we dubbed "SpamSystem.NET"[5]. The application provided both full email functionality with Gmail accounts and used our own customizable kNN email classification scheme for Spam-filtration.

To achieve email client functionality, we relied on some aspects of the MailSystem.NET open source libraries[6]. MailSystem.NET provided functionality that allowed us to easily open connections to Gmail's servers. For our purposes, we modified the Gmail account settings so that Gmail automatically sent all messages to the inbox, even those messages identified as Spam by Google's filters. This allowed us to test our own implementation without outside interference. It is worth noting that this is not necessary, and both filters could be used in tandem. We did this strictly for the purpose of testing. MailSystem.NET's libraries allowed us to send and receive email messages, create our own functions to manage the user's Gmail account, maintain bidirectional synchronization with Google's servers). It also allowed the system to detect changes to the user's account, which happen outside of our application. The main user interface can be seen in Fig. 3, with the user's different email folders and options for standard email client actions. MailSystem.NET also provided functionality for creating email objects, which allowed us to more easily parse emails for the classification metrics we required.

Additionally, our application provides easy options for managing the kNN Spam-filter. Fig. 4 shows the options-screen of our SpamSystem.NET application. In addition to managing the user's email account, there are options for maintaining a Black-List and a White-List, which specify keywords, phrases, domain names, and email addresses that, should they appear in an email, indicate that the email should automatically go to the Spam folder (for Black- List items) or the inbox (for White-List items). There are also necessary features for maintaining the kNN Spam-filter itself. There are options for determining the time interval between training-set rebasings, as well as options for determining which GPU(s) should be used to run the kNN classification, should more than one GPU be present in the system.

To summarize how the system operates, first an unclassified incoming email is received from the Gmail server in the form of an IMAP message. It is then parsed by our application into quantifiable attributes. Next, it is sent to the kNN classifier, which determines whether it is Ham or Spam. Finally, the email is moved to the appropriate email folder in our application, and the application notifies the Gmail server to do the same. A detailed diagram representing the components and how they interact can be seen in Fig. 2.

Our system runs locally on the user's machine. Our goal in developing this local implementation was to demonstrate its efficiency and scalability.

[5]For this implementation stage, we gratefully acknowledge the contribution of our student colleague Leonard Patterson.

[6]http://mailsystem.codeplex.com/

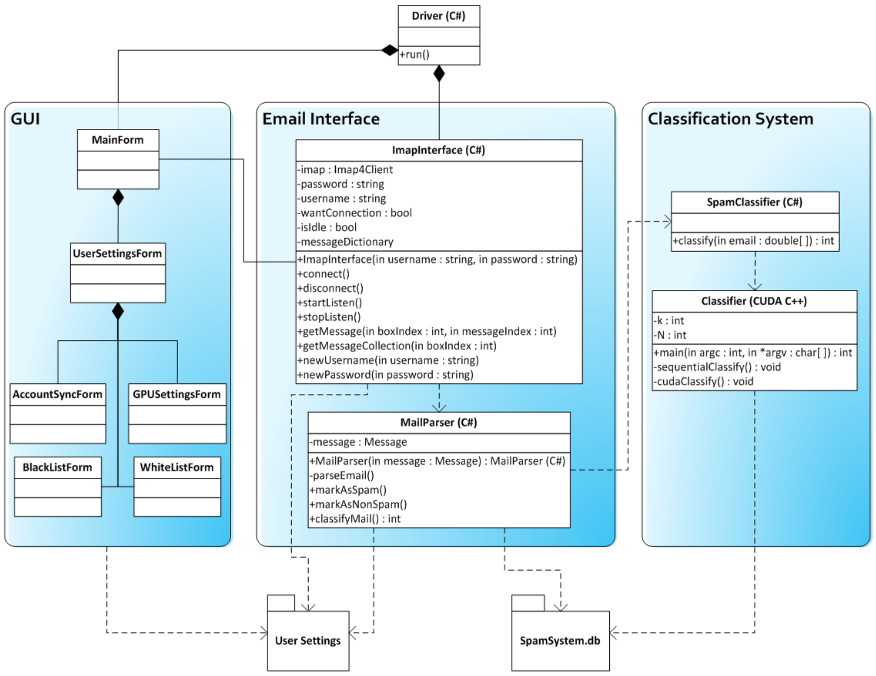
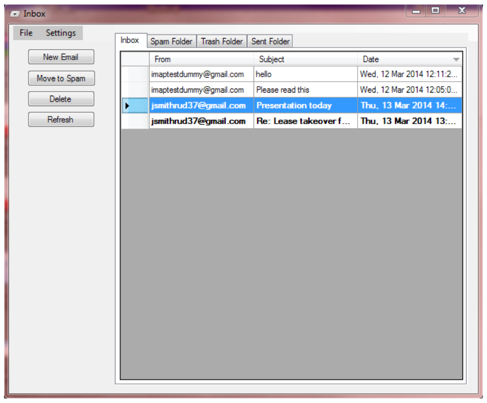
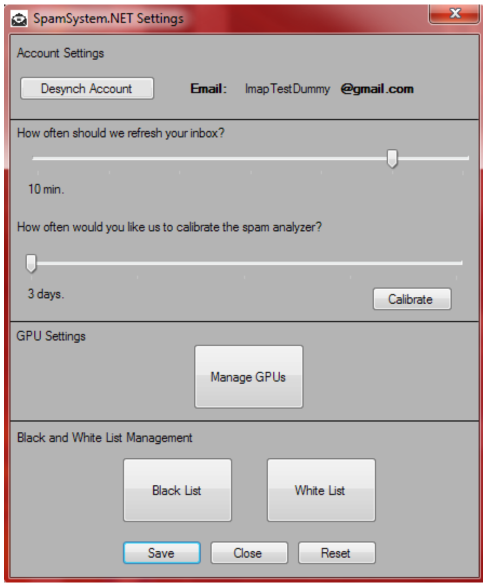


Figure 2: The overall system architecture for SpamSystem.NET.



From	Subject	Date
imaptestdummy@gmail.com	hello	Wed, 12 Mar 2014 12:11:2...
imaptestdummy@gmail.com	Please read this	Wed, 12 Mar 2014 12:05:0...
jsmithrud37@gmail.com	Presentation today	Thu, 13 Mar 2014 14:...
jsmithrud37@gmail.com	Re: Lease takeover f...	Thu, 13 Mar 2014 13:...

Figure 3: Main screen of SpamSystem.NET's user interface.



Email: ImapTestDummy @gmail.com

How often should we refresh your inbox?

10 min.

How often would you like us to calibrate the spam analyzer?

3 days.

Figure 4: Option screen of SpamSystem.NET's user interface.

5 Experiments

A significant amount of previous research has been done showing that the kNN algorithm can be highly accurate (see, for example, (Firte, Lemnaru, and Potolea 2010)). The algorithm's real deficit lies in the cost of its computation. As a result, the primary focus of our research and resulting benchmarks relate to demonstrating the efficiency gained by implementing kNN in CUDA. That being said, one cannot discuss a classification algorithm without discussing accuracy. This section details both time and accuracy benchmarks.

Our implementation was designed and tested on a GeForce GTX 260 (PCIe 2.0) on a Windows 7 PC with an Intel i7 950 CPU. All of our tests were run on this configuration.

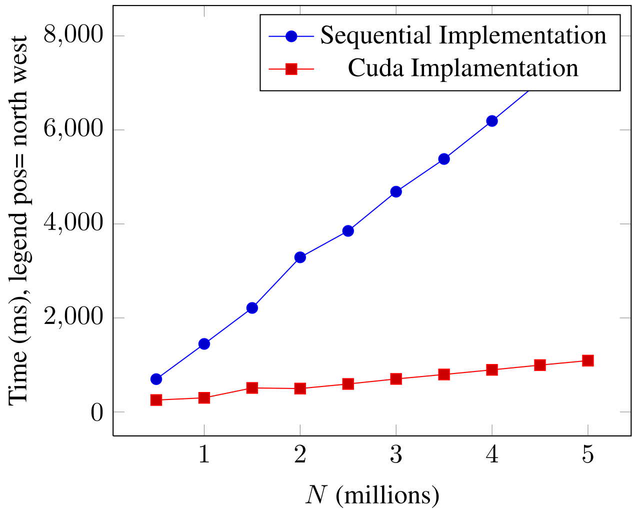


Figure 5: Performance of CUDA implementation on GTX GPU versus Sequential implementation on CPU.

Efficiency Benchmarks

All efficiency benchmarks are the result of 10 trials on a dummy dataset.

Fig. 5 shows the efficiency in terms of time as the size of the training set (N) increases. Here we compare a naïve, sequential implementation of kNN versus our implementation on the GTX 260. For these tests we used a k value of 10.

Fig. 6 shows the speedup gained by the CUDA implementation on the GTX 260 over the sequential implementation as N increases.

Accuracy Benchmarks

All accuracy benchmarks are the result of a 10-fold cross-validation (10:90 test-set to training-set ratio) on the SpamAssassin dataset.

Fig. 7 shows the accuracy (in terms of false-negatives and false positives) for varying threshold values (θ). For this experiment, a k-value of 10 was used. The results show that while higher θ-values yield fewer instances of false-positive classifications, it also yields more instances of false-negative classifications (as one would expect). Depending on the environment in which this application was being used, a proper θ could be easily determined via training to fit the desired false-positive to false-negative ratio, but a "best" value to use is not necessarily apparent.

Fig. 8 shows the accuracy (in terms of false-negatives and false positives) for varying values of k. For this experiment, a θ-value of 0.7 was used. Based on our results, we determined that $k = 3$ was optimal for our purposes based on its false-positive to false-negative ratio. Additionally, since smaller k-values tend to yield faster computational performance, this choice was obvious. That being said, an optimal k value could also be determined via training for any environment using this application based on the specific needs of that environment.

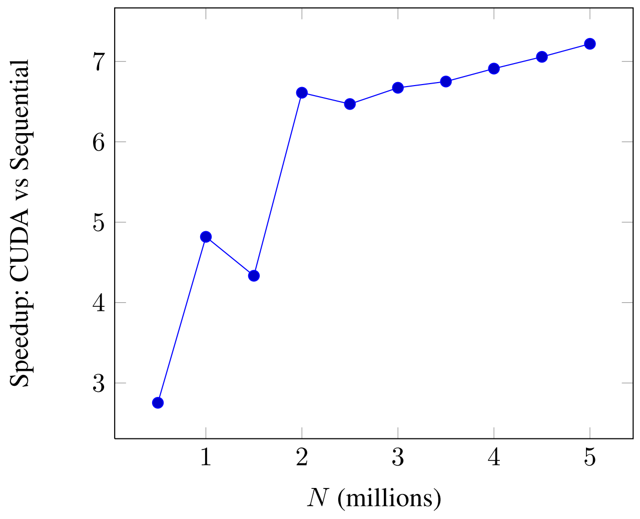


Figure 6: Speedup of CUDA implementation on GTX GPU versus Sequential implementation on CPU.

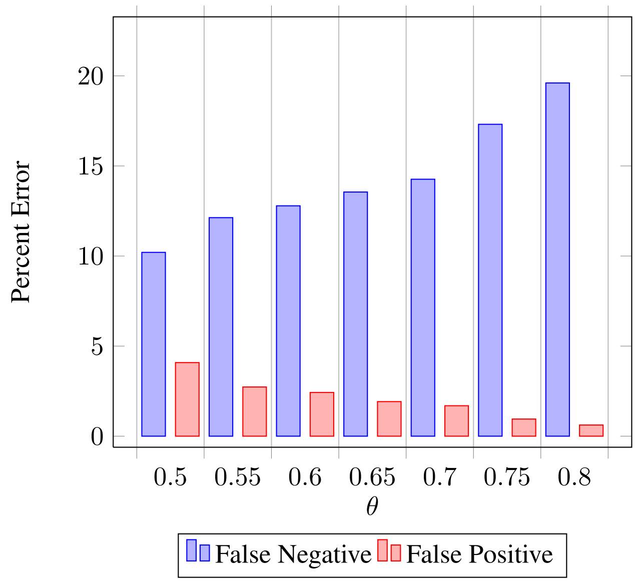


Figure 7: Accuracy of classification on varying θ values.

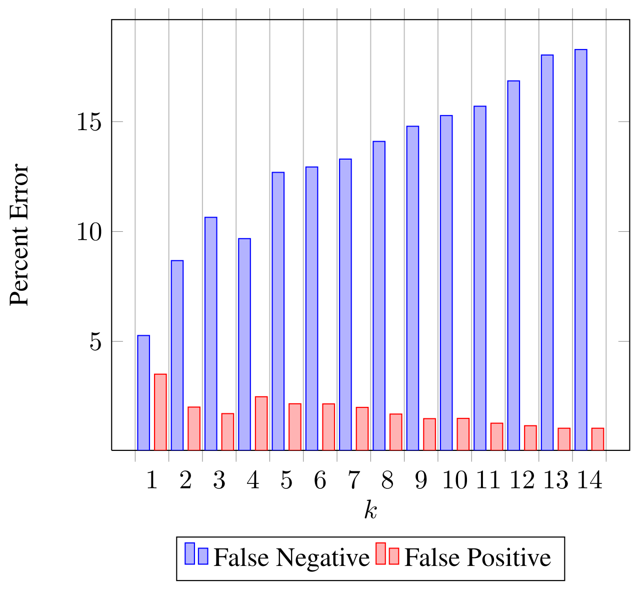


Figure 8: Accuracy of classification on varying k values.

6 Conclusions

Although many email Spam-filtering tools exists in the world, due to the existence of spammers and adoption of new techniques, email Spam-filtering becomes a challenging problem to researchers. Automatic email filtering seems to be the most effective method for countering Spam at the moment and a tight competition between spammers and spam-filtering methods is going on: as anti-Spam methods become more refined, so too do those of the spammers (Tretyakov 2004).

In this work, we have demonstrated an efficient implementation of the kNN algorithm utilizing CUDA-enabled GPUs. Under our scheme, an incoming email is parsed into its quantifiable attributes and is then sent to the classification system. Here, the distances between the newly-processed email and each email in the training set are quickly calculated on the GPU. Afterwards, a parallel reduction process is used to identify the k training set emails with the smallest distances from the query email. These k email cast weighted votes for their own classification. If the Spam-vote exceeds the required threshold (θ), then the query email is classified as Spam. Otherwise, it is classified as Ham. Our experimental results show that our implementation is highly efficient and scalable. This efficiency, coupled with kNN's previously demonstrated accuracy for classification, makes it potentially beneficial for large-scale Spam-filtration, and could be easily adapted to tackle other similar problems.

References

Arefin, A. S.; Riveros, C.; Berretta, R.; and Moscato, P. 2012. GPU-FS-kNN: A software tool for fast and scalable kNN computation using GPUs. *PLoS one* 7(8):e44000.

Blanzieri, E., and Bryl, A. 2008. A survey of learning-based techniques of email spam filtering. *Artif. Intell. Rev.* 29(1):63–92.

Cormack, G. V. 2008. Email spam filtering: A systematic review. *Found. Trends Inf. Retr.* 1(4):335–455.

Firte, L.; Lemnaru, C.; and Potolea, R. 2010. Spam detection filter using kNN algorithm and resampling. In *Intelligent Computer Communication and Processing (ICCP), 2010 IEEE International Conference on*, 27–33.

Lopes, L., and Ribeiro, B. 2011. GPUMLib: An efficient open-source GPU machine learning library. *International Journal of Computer Information Systems and Industrial Management Applications* 2150–7988.

Panigrahi, P. 2012. A comparative study of supervised machine learning techniques for spam e-mail filtering. In *Computational Intelligence and Communication Networks (CICN), 2012 Fourth International Conference on*, 506–512.

Sallab, A. A. A., and Rashwan, M. A. 2012. E-mail classification using deep networks. *Journal of Theoretical and Applied Information Technology* 37:241–251.

Tretyakov, K. 2004. Machine learning techniques in spam filtering. Technical report, Institute of Computer Science, University of Tartu.

Yokoyama, T.; Ishikawa, Y.; and Suzuki, Y. 2012. Processing all k-nearest neighbor queries in hadoop. In Gao, H.; Lim, L.; Wang, W.; Li, C.; and Chen, L., eds., *Web-Age Information Management*, volume 7418 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg. 346–351.