GPU Acceleration of Document Similarity Measures for Automated Bug Triaging

Tim Dunn[1,2], Natasha Kholgade Banerjee[2], and Sean Banerjee[1]

[1]Department of Electrical and Computer Engineering, Clarkson University, Potsdam, NY, USA
[2]Department of Computer Science, Clarkson University Potsdam, NY, USA
{dunntj, nbanerje, sbanerje} @clarkson.edu

***Abstract*—Large-scale open source software bug repositories from companies such as Mozilla, RedHat, Novell and Eclipse have enabled researchers to develop automated solutions to bug triaging problems such as bug classification, duplicate classification and developer assignment. However, despite the repositories containing millions of usable reports, researchers utilize only a small fraction of the data. A major reason for this is the polynomial time and cost associated with making comparisons to all prior reports. Graphics processing units (GPUs) with several thousand cores have been used to accelerate algorithms in several domains, such as computer graphics, computer vision and linguistics. However, they have remained unexplored in the area of bug triaging. In this paper, we demonstrate that the problem of comparing a bug report to all prior reports is an embarassingly parallel problem, that can be accelerated using graphics processing unit (GPUs). Comparing the similarity of two bug reports can be performed using frequency based methods (e.g. cosine similarity and BM25F), sequence based methods (e.g. longest common substring and longest common subsequence) or topic modeling. For the purpose of this paper we focus on cosine similarity, longest common substring and longest common subsequence. Using an NVIDIA Tesla K40 GPU, we show that frequency and sequence based similarity measures are accelerated by 89 and 85 times respectively when compared to a pure CPU based implementation. Thus, allowing us to generate similarity scores for the entire Eclipse repository, consisting of $498,161$ reports in under a day, as opposed to 83.4 days using a CPU based approach.**

I. Introduction

Nearly all large-scale software projects contain flaws or defects, otherwise known as software bugs. Bug tracking systems, such as Bugzilla and JIRA, allow both developers and users to submit textual bug reports on observed failures. Bugzilla is currently being used by companies such as RedHat, Mozilla, Eclipse and Novell [1]. As shown in Table I, such repositories already contain millions of bug reports and are receiving several hundred new reports each day. As new bug report comes in, the development team must determine if the issue at hand is an actual problem, an issue that cannot be replicated (WORKSFORME), an issue that will not be fixed as it is beyond the scope of the project (WONTFIX), an issue that does not have sufficient information (INCOMPLETE), an issue describing a problem with a different system (INVALID), or simply a duplicate of an existing bug report (DUPLICATE).

Open source bug tracking systems, such as Bugzilla, have provided researchers with rich datasets from a variety of software applications. The past decade has seen the emergence of research tracks in assigning new or duplicate labels to each bug report (bug classification), identifying the original report associated with each duplicate report (duplicate classification), and determining the developer most qualified to fix the problem (developer assignment). For the scope of this paper, we define bug triaging as either the manual or automated process associated with bug classification, duplicate detection and developer assignment. Even though the datasets illustrated in Table I contain millions of problem reports, most researchers continue to use small subsets of the data as shown in Table II. The foremost reason for this is the time and cost associated with mining large-scale repositories [8]. The complexity of mining large-scale repositories arises due to the fact that each bug report must be compared to all prior reports. For a given repository of size n, the total number of comparisons needed is given by $1+2+3+4+...+n-1$ amounting to $O(n^2)$. For a repository such as Mozilla, approximately 8×10^{11} comparisons must be made if a researcher is interested in generating similarity scores for all reports within the repository.

The emergence of general purpose graphics processing units or GPUs with several thousand processing cores has enabled the acceleration of scientific computing algorithms by several orders of magnitude. For example, the NVIDIA Tesla K40 GPU contains 2880 cores and 12GB of memory [2]. Originally introduced to accelerate rendering algorithms in computer graphics [26], the GPU has now extended into several branches of computer science such as computer vision [7], [21], molecular modeling [29], and linguistics [14]. However, it has remained unexplored in the domain of bug triaging where each bug report is compared to all prior reports using frequency-based (e.g. cosine similarity or BM25F), sequence-based (e.g. longest common subsequence or substring) or topic modeling methods. The problem of comparing each bug report to all prior reports is an embarrassingly parallel problem, as the comparisons can be made independent of each other,

Repository	Reports	Reports per day
RedHat	$1,357,998$	210
Mozilla	$1,287,896$	197
Novell	$989,610$	233
Eclipse	$498,161$	92

TABLE I
Size of Bugzilla Repositories (retrieved on: July 19, 2016

thus making bug report matching a candidate for GPU-based acceleration.

In this paper we present our analysis on the effects of accelerating three common document similarity metrics, namely cosine similarity, longest common substring and longest common subsequence, using an NVIDIA Tesla K40 GPU. Unlike domains with static data sizes, each problem report consists of a variable document length. As a result, we apply a data driven approach for selecting a static array size for the GPU. Our framework is tested on the Eclipse repository, and provides a speed up of $85\times$ for longest common substring and subsequence, and $89\times$ for cosine similarity when compared to a pure CPU based implementation. Our approach can generate similarity scores for all reports within the Eclipse repository using three similarity metrics in under 24 hours.

The remainder of the paper is organized as follows. In Section II we discuss the related work in open source bug tracking systems, as well as emerging research using GPUs in related fields. In Section III, we discuss the implementation details used for developing the accelerated versions of the algorithms. Section IV presents the results of accelerating the similarity measures and provide comparisons to a purely CPU based approach. We provide validity threats in Section V and conclude the paper in Section VI with potential directions for future work.

II. Related Work

A. Bug Triaging

Research in automated bug triaging has focused on determining if a report is new or duplicate [11], [16], [18], determining the original report associated with each known duplicate [18], [33], [31], [30], [32], [25], [27], [10], [9], [20] and determining the developer most qualified to fix a problem [28], [12], [24], [5], [15], [6] using a variety of frequency, sequence and topic modeling based methods. Each contribution to the domain has been strictly performed using a CPU based approach, or a cluster based approach [11].

Dataset	Range	Recall
Eclipse	Jan 2008 - Dec 2008	68% [31]
Eclipse	Start - Dec 2009	46% [32]
Eclipse	Jan 2008 - Dec 2008	78% [30]
Eclipse	Start - Dec 2007	71% [30]
Firefox	Apr 2004 - Jun 2004	93% [33]
Firefox	Apr 2002 - Jul 2007	53% [31]
Firefox	Apr 2002 - Jul 2007	70% [31]
Firefox	Start - Jun 2010	53% [27]
Firefox	Start - Mar 2012	68% [9]
Firefox	Start - Sept 2005	50% [18]
Mozilla	Jan 2010 - Dec 2010	68% [30]
Mozilla	Feb 2005 - Oct 2005	51% [20]
Eclipse	Jan 2002 - Sept 2002	30% [15]
Eclipse	Sept 2004 - May 2005	64% [6]
Firefox	Sept 2004 - May 2005	57% [6]
Eclipse	Oct 2005 - May 2006	70% [5]
Firefox	Feb 2006 - Sept 2006	75% [5]
Eclipse	Apr 2001 - Nov 2008	71% [24]
Eclipse	Oct 2001 - Mar 2010	86% [12]
Mozilla	May 1998 - Mar 2010	84% [12]
Eclipse	Dec 2006 - Jan 2007	89% [28]
Mozilla	Dec 2006 - Jan 2007	59% [28]

TABLE II
SIZE OF REPOSITORIES BEING USED IN BUG TRIAGING RESEARCH

B. GPU Acceleration

GPUs have traditionally been used to accelerate rendering algorithms in computer graphics [26]. General purpose GPUs have been used in the past decade to accelerate algorithms on point-level processing in computer vision [7], molecular modeling [29], convolutional neural networks for object detection and recognition [21], and deep neural networks for speech recognition [17]. To the best of our knowledge, there are no approaches on accelerating bug triaging using GPUs. Approaches most allied to our work that perform GPU-based acceleration fall in the area of text mining and document retrieval. Zhang and Cui [35] provide an approach to perform term frequency based comparisons. Their performs over-subscription of tasks to the GPU cores to ensure high utilization. Unlike our approach, they do not normalize for document size. Kysenko et al. [23] perform non-negative matrix factorization to cluster documents using a GPU. In [34], the authors perform the initial IO-bound operations of text mining approaches using a distributed framework on CPUs, and implement further compute-bound operations on distributed GPUs. In [19], the authors use the Bloom filter [13] to perform document retrieval comparisons. In general, the problem of performing document comparisons on a GPU is rendered challenging due to the fact that document sizes may not be known *a priori*. In this work, we determine the optimal container size to store the problem report data on the GPU as the quantity that simultaneously optimizes the shared memory utilization, GPU multiprocessor occupancy, and number of reports processed at a given time. We use a data-driven approach to validate the optimal container size.

While the authors of [22] provide an approach to accelerate the longest common subsequence approach used in our work for DNA matching in computational genomics, their approach performs comparisons on the four building blocks of the DNA alphabet, i.e., the bases A, G, C, and T. The problem reports used in this work have up to 67 building blocks, consisting of the 26 letters converted to lower case, 10 numerals, and up 31 non-alphanumeric symbols occuring on an English language keyboard. The higher number of building blocks renders the character level comparisons for word matching in longest common subsequence computations far more time-intensive. Additionally, we are the first to implement the longest common substring approach on a GPU for automated bug triaging.

III. Framework

Our GPU accelerated framework was developed using an NVIDIA Tesla K40 GPU consisting of 2880 cores and 12GB of memory installed on a 10 core Intel powered Asus ESC4000-G3 server with 128GB of RAM.

A. Data Preprocessing

The Eclipse repository chosen for our study consists of all bug reports submitted from the start of the project until

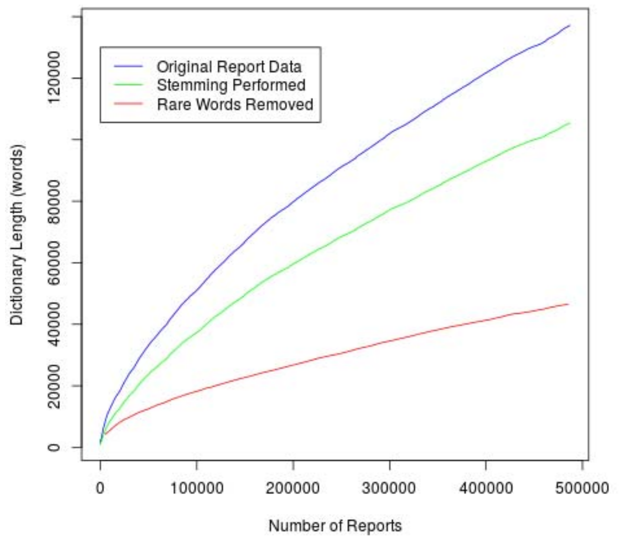

Fig. 1. Effect of stemming and rare word removal on dictionary size. Application of stemming and rare word removal reduces the size of the dictionary by one third.

February 5, 2016 and comprises of $487,119$ reports. We first sanitize the data by applying tokenization, stop word removal, and stemming (using the Porter stemming algorithm [3]). Next, we generate a dictionary of unique words contained within the repository and remove any words that occurred less than 3 times. These words consist of misspellings, garbage words such as "aaaaaaaaaaa", or words transliterated from a foreign language. This step reduces the dictionary size to increase efficiency of the computations in Subsection III-B, and reduces noise. Figure 1, below, shows the effects of stemming and rare word removal on the final size of the dictionary.

Once the final dictionary is generated, we replace each word with a unique 2-byte integer identifier corresponding to the index of the word in the sorted dictionary. This ensures that each word is the same length, and allows for a lossless data compression ratio of $2.49 : 1$, decreasing the total size of all reports from 263 MB to 106 MB.

B. Allocating Static Array Size

GPU memory requires allocation of constant size arrays. While the data compression in Section III-A converts each word to a fixed length, each report contains a variable number of words. We refer to the number of words in a report as the description length. The optimal choice of maximum description length for initializing the arrays corresponds to one that minimizes information loss while maximizing the utilization of memory space and the number of reports processed concurrently. The Tesla K40 GPU consists of 15 multiprocessors, each of which contains 192 CUDA cores. The CUDA program is partitioned into multithreaded blocks. Threads in a single block execute simultaneously on a single multiprocessor. Each multiprocessor can execute upto a maximum of 2048 threads, while each block can be allocated a maximum of 1024 threads. Additionally, each multiprocessor has 49152 bytes of shared memory. Each thread in our approach computes the match between two reports. We allocate n^2 threads to each block, where n reports are compared to n other reports. This leads to there being $2n$ reports on each block. Suppose the maximum description length is N. Then since each word has a 2-byte identifier, the maximum number of bytes available per report is $2N$ bytes. Simultaneously maximizing the use of shared memory, the thread utilization (or occupancy) of each multiprocessor, and the number of reports processed at a single time requires that

$$49152 \, \frac{\text{bytes}}{\text{processor}} = 2 \, \frac{\text{bytes}}{\text{word}} \times N \, \frac{\text{words}}{\text{report}} \times 2n \, \frac{\text{reports}}{\text{block}} \times \frac{1 \text{ block}}{n^2 \text{ threads}} \times 2048 \, \frac{\text{threads}}{\text{processor}}, \text{ or,} \tag{1}$$

$$N = 6n. \tag{2}$$

Since the maximum number of threads per block for the Tesla K40, i.e., the maximum value of n^2, is 1024 or 32×32, we analyzed the effect of using $n^2 = 8 \times 8$, 16×16, and 32×32 threads per block, amounting to $2n = 16$, 32, and 64 reports respectively. With 8 reports, the number of threads per block is 64, requiring 32 blocks to maximally utilize all 2048 threads of the multiprocessor. However, the Tesla K40 can run a maximum of 16 blocks at a single time. With 32×32 threads or 64 reports per block, 2 blocks run on the same multiprocessor at a single time, leading to a maximum of 64×2 or 128 reports being worked on by a single multiprocessor at a given time instant. However, with 16×16 threads or 32 reports per block, 8 blocks run on the same multiprocessor, leading to a maximum of 32×8 or 256 reports being processed at a single time instant. Therefore, using 16×16 threads or 32 reports maximizes the occupancy, i.e., uses all 2048 threads per multiprocessor by running 8 blocks at a time, and maximizes the number of reports processed, while simultaneously filling 49152 bytes of shared memory. At $n = 16$, N turns out to be 96 words per report according to Equation (2). We use the last word as a placeholder for thread metadata, yielding a maximum of 95 words per report.

We apply a data-driven approach to validate the optimal array size by determining the distribution of description lengths within the repository as shown in Figure 2. The bar chart at the top of Figure 2 shows the frequency of reports versus report description length. The graph at the bottom of Figure 2 shows the percentage of reports shorter than the description length indicated by the x-axis. Both plots in Figure 2 share the x-axis. The bar chart demonstrates the typical heavy-tailed distribution expected of large repositories where a larger percentage of reports have smaller lengths. This observation is also borne out by the asymptotic rise of the curve at the bottom of Figure 2. Figure 2 shows that 84.7% of reports are shorter than 95 words, i.e., that only 15.3% of all reports are affected by this choice of report length. Additionally, reports consisting of more than 95 words generally contain execution trace information which may not be found in all reports within

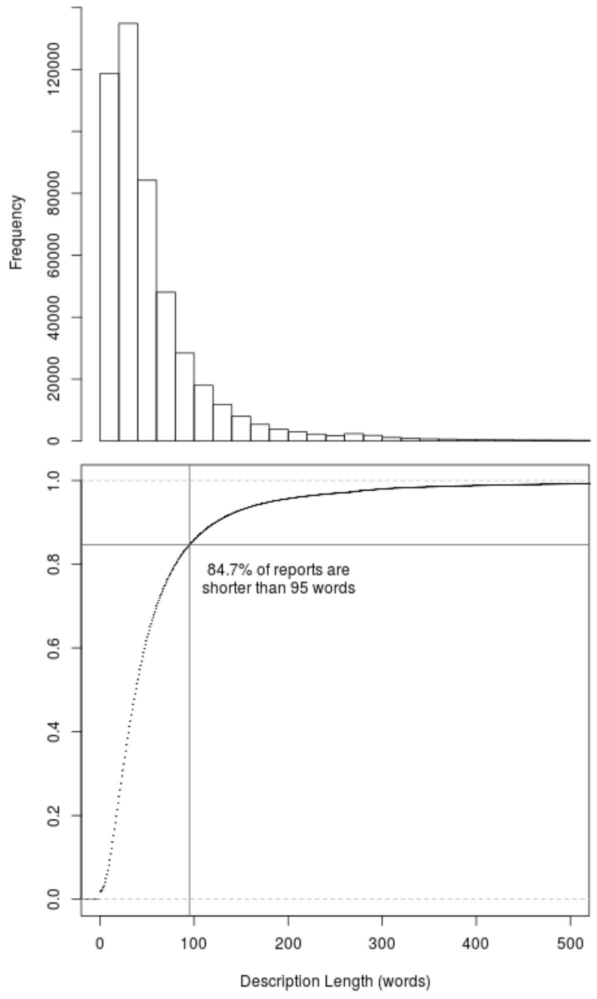

Fig. 2. The top graph shows the Eclipse description length distribution as a function of the report size. From the bottom graph we note that 84.7% of the reports within Eclipse have 95 words or less.

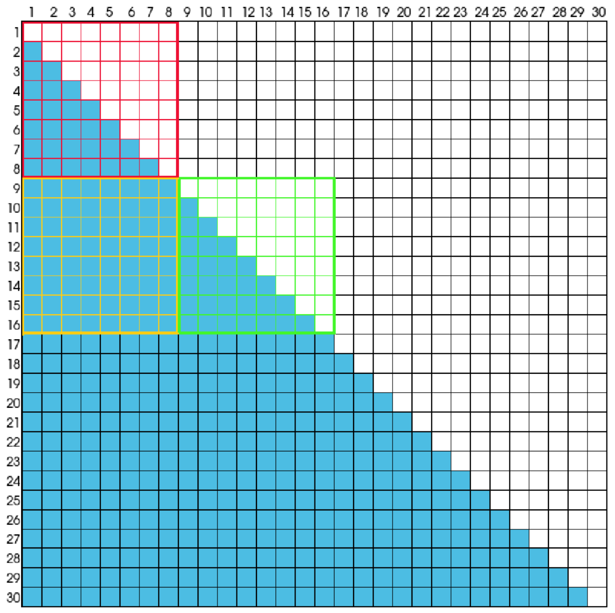

Fig. 3. Secondary tiling for comparing more than $30,000$ reports. Secondary tiling is needed to prevent the GPU global memory from being exhausted.

Repository	Reports	Comparisons Needed	Runtime
Eclipse	$1,000$	$500,500$	29 seconds
Eclipse	$10,000$	$50,005,000$	48 minutes
Eclipse	$20,000$	$200,010,000$	3.2 hours
Eclipse	$498,161$	1.24×10^{11}	83.4 days
Mozilla	$1,287,896$	8.29×10^{11}	557.7 days

TABLE III
BASELINE RUNTIMES FOR USING A SINGLE CPU CORE.

the repository and are hence excluded from the similarity metrics.

We use the traditional tiled matrix approach to implement the GPU accelerated algorithms. Since $O(n^2)$ comparisons are made for each comparison method, the output exhausts the global memory of the GPU when analyzing more than $30,000$ reports at a time using a single layer of tiling. To address this issue, we add a secondary layer of tiling such that each kernel compares 8192 reports to 8192 other reports before transferring data back to the system memory and beginning another kernel. Although this method requires additional data transfers, it is able to handle any number of comparisons without exhausting the GPU memory. Figure 3 illustrates this concept with kernels of size 8×8, where the colored grids represent the comparisons made by the first three kernels.

IV. GPU ACCELERATION RESULTS

a) Baseline CPU Results: To perform a fair comparison of our GPU accelerated algorithms against a traditional CPU implementation, we implemented versions of CPU algorithms for cosine similarity, longest common subsequence, and longest common substring from [27] and [9], where strings were replaced by 2-byte integers. We designed each algorithm to be purely sequential, i.e., to use a single CPU core. Baseline runtimes for the CPU-based sequential algorithms were obtained using an Asus ESC4000-G3 server with an Intel Xeon E5-2660-v3 10 core processor and 128GB of RAM. Each algorithm was run on on core of the 10 core processor. As shown by Table III, we obtained runtimes for the first $1,000$, $10,000$ and $20,000$ reports from the Eclipse repository and extrapolated to the complete Eclipse and Mozilla repository. Researchers interested in generating similarity scores for the entire Eclipse and Mozilla repository using the three methods would require 83.4 days and 1.5 years respectively. While a cluster based or cloud based approach can expedite this process, the monetary cost and data transfer latency far outweigh the benefits of performing computations on the cloud [8].

b) Comparison between GPU and CPU-based Approaches: In Figure 4 we compare the runtimes of the CPU-based sequential version and GPU-based parallel version of cosine similarity. For the sequential approach we provide runtimes for the first $20,000$ reports to illustrate the computational overhead. We provide runtimes for the first $100,000$ reports

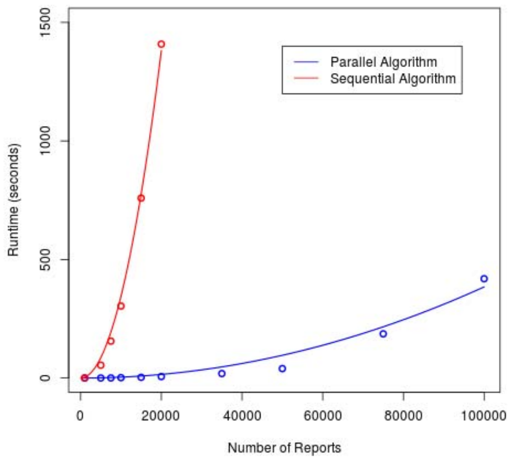

Fig. 4. Runtimes of the GPU-based parallel algorithm and CPU-based sequential algorithm for cosine similarity.

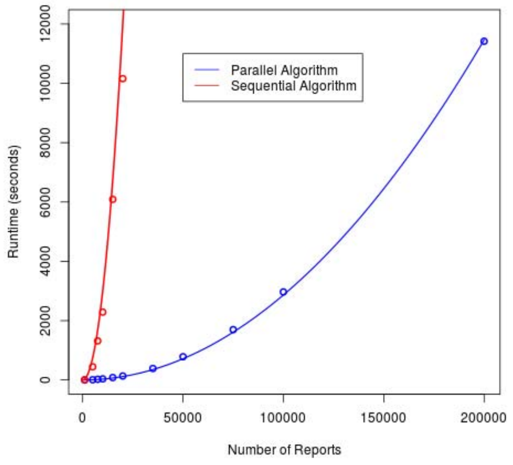

Fig. 5. Runtimes of the GPU-based parallel algorithm and CPU-based sequential algorithm for longest common subsequence and substring.

using the parallel approach. Additionally, we fit a regression line to the sequential and parallel algorithms. The GPU-based parallel cosine similarity approach provides a speed up of $89\times$ when compared to the CPU-based sequential version.

In Figure 5 we compare the runtimes for the CPU-based sequential and GPU-based parallel algorithms for the longest common subsequence and substring approaches. The longest common subsequence and substring can be computed in parallel when traversing through the two documents. We provide runtimes for the first $20,000$ reports using the sequential approach, and the first $200,000$ reports using the parallel approach. The GPU-based parallel algorithm provides an $85\times$ speed up when compared to the CPU-based sequential version.

The acceleration achieved for cosine similarity is significantly higher than both the longest common subsequence and substring algorithms. This is expected as the dot product between the two reports is a computationally inexpensive operation. The longest common substring and subsequence algorithms utilize dynamic programming and perform $O(mn)$ operations to compare two reports of lengths m and n.

c) Projected Results: As shown in Figure 6, we determine the total time required to generate similarity match scores using all three methods by fitting polynomial regression lines to both the sequential and parallel similarity algorithms. The GPU accelerated approach can generate similarity matches for the entire Eclipse repository in under 24 hours. In Table IV we extend the projection to the Mozilla dataset and determine that the sequential approach will take 557.7 days, while the parallel approach completes in 6.5 days.

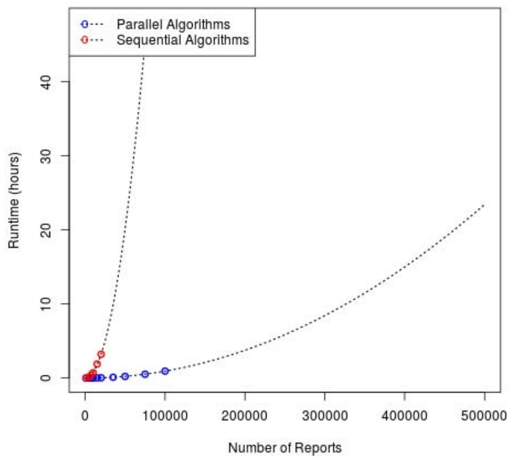

Fig. 6. Computational time projected to compute similarity scores using all three methods on the complete Eclipse dataset

Repository	Total Reports	CPU single core	GPU
Eclipse	$498,161$	83.4 days	23.3 hours
Mozilla	$1,287,896$	557.7 days	6.5 days

TABLE IV
PROJECTED RESULTS FOR ECLIPSE AND MOZILLA REPOSITORY.

V. Threats to Validity

Our current accelerated approach has been tested on the Eclipse dataset, in future we will test the approach on larger datasets from Mozilla, RedHat and Novell to reduce potential external validity threats. The chosen document length threshold may not be applicable to other datasets, in future we will will explore the distribution of document sizes in datasets from Mozilla, RedHat and Novell. Finally, the Tesla K40 GPU chosen for this study has 2880 cores and a maximum memory capacity of 12GB. The acceleration numbers reported may not be applicable to other GPUs. However, the ease of procuring a Tesla K40 through the NVIDIA Hardware Grant program will enable replication of our study [4].

VI. Conclusions and Future Work

Through an empirical study we demonstrate that existing CPU based implementations of common similarity metrics, such as cosine similarity, longest common subsequence and substring, are inadequate when processing on bug repositories with millions of reports. The time required to generate match scores can take in the order of months or years. Using GPU based implementations, we provide a speedup of $85\times$ for sequence based methods and $89\times$ for frequency based methods. This reduces the computation time from months and years, to hours and several days. The task of automated bug triaging is far from solved, by accelerating the process, we can apply advanced fusion techniques that combine multiple similarity measures in a matter of days as opposed to months. Our GPU based approach can enable researchers to mine complete repositories as opposed to small subsets. Further, the cost of an individual GPU is far less than a cluster or cloud based system.

In future we will develop GPU accelerated topic modeling techniques and provide a complete suite of similarity metrics. Additionally, we are currently in the process of obtaining full repositories from Mozilla, RedHat and Novell to test the effectiveness of our approach on much larger repositories. Furthermore, we have recently obtained additional GPUs to enable further parallelization and investigations on effectively transferring data across GPUs.

Acknowledgments

We gratefully acknowledge the support of NVIDIA Corporation with the donation of the Tesla K40 GPU used for this research, and Corning Inc. for the STEM research stipend.

References

[1] https://www.bugzilla.org/installation-list/.

[2] http://www.nvidia.com/object/tesla-servers.html/.

[3] http://www.tartarus.org/-martin/PorterStemmer.

[4] https://developer.nvidia.com/academic_gpu_seeding/.

[5] J. Anvik. *Assisting bug report triage through recommendation*. PhD thesis, University of British Columbia, 2007.

[6] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *Proceedings of the 28th international conference on Software engineering*, pages 361–370. ACM, 2006.

[7] P. Babenko and M. Shah. Mingpu: a minimum gpu library for computer vision. *Journal of Real-Time Image Processing*, 3(4):255–268, 2008.

[8] S. Banerjee and B. Cukic. On the cost of mining very large open source repositories. In *Proceedings of the First International Workshop on BIG Data Software Engineering*, pages 37–43. IEEE Press, 2015.

[9] S. Banerjee, B. Cukic, and D. Adjeroh. Automated duplicate bug report classification using subsequence matching. In *High-Assurance Systems Engineering (HASE), 2012 IEEE 14th International Symposium on*, pages 74–81. IEEE, 2012.

[10] S. Banerjee, Z. Syed, J. Helmick, and B. Cukic. A fusion approach for classifying duplicate problem reports. In *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on*, pages 208–217. IEEE, 2013.

[11] S. K. Banerjee. *An automated framework for problem report triage in large-scale open source problem repositories*. PhD thesis, West Virginia University, 2014.

[12] P. Bhattacharya, I. Neamtiu, and C. R. Shelton. Automated, highly-accurate, bug assignment using machine learning and tossing graphs. *Journal of Systems and Software*, 85(10):2275–2292, 2012.

[13] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[14] J. Canny, D. Hall, and D. Klein. A multi-teraflop constituency parser using gpus. *Architecture*, 3:3–5.

[15] D. Čubranić. Automatic bug triage using text categorization. In *In SEKE 2004: Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering*. Citeseer, 2004.

[16] L. Feng, L. Song, C. Sha, and X. Gong. Practical duplicate bug reports detection in a large web-based development community. In *Web Technologies and Applications*, pages 709–720. Springer, 2013.

[17] A. Hannun, C. Case, J. Casper, B. Catanzaro, G. Diamos, E. Elsen, R. Prenger, S. Satheesh, S. Sengupta, A. Coates, et al. Deep speech: Scaling up end-to-end speech recognition. *arXiv preprint arXiv:1412.5567*, 2014.

[18] L. Hiew. *Assisted detection of duplicate bug reports*. PhD thesis, The University Of British Columbia, 2006.

[19] A. Iacob, L. Itu, L. Sasu, F. Moldoveanu, and C. Suciu. Gpu accelerated information retrieval using bloom filters. In *System Theory, Control and Computing (ICSTCC), 2015 19th International Conference on*, pages 872–876, 2015.

[20] N. Jalbert and W. Weimer. Automated duplicate detection for bug tracking systems. In *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, pages 52–61. IEEE, 2008.

[21] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. B. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. *CoRR*, abs/1408.5093, 2014.

[22] K. Kawanami and N. Fujimoto. Facing the multicore-challenge ii. chapter GPU Accelerated Computation of the Longest Common Subsequence, pages 84–95. 2012.

[23] V. Kysenko, K. Rupp, O. Marchenko, S. Selberherr, and A. Anisimov. Gpu-accelerated non-negative matrix factorization for text mining. In *International Conference on Application of Natural Language to Information Systems*, pages 158–163. Springer, 2012.

[24] D. Matter, A. Kuhn, and O. Nierstrasz. Assigning bug reports using a vocabulary-based expertise model of developers. In *2009 6th IEEE International Working Conference on Mining Software Repositories*, pages 131–140. IEEE, 2009.

[25] A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, D. Lo, and C. Sun. Duplicate bug report detection with a combination of information retrieval and topic modeling. In *Automated Software Engineering (ASE), 2012 Proceedings of the 27th IEEE/ACM International Conference on*, pages 70–79. IEEE, 2012.

[26] H. Nguyen. *Gpu Gems 3*. Addison-Wesley Professional, first edition, 2007.

[27] T. Prifti, S. Banerjee, and B. Cukic. Detecting bug duplicate reports through local references. In *Proceedings of the 7th International Conference on Predictive Models in Software Engineering*, page 8. ACM, 2011.

[28] R. Shokripour, J. Anvik, Z. M. Kasirun, and S. Zamani. Why so complicated? simple term filtering and weighting for location-based bug report assignment recommendation. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 2–11. IEEE Press, 2013.

[29] J. E. Stone, J. C. Phillips, P. L. Freddolino, D. J. Hardy, L. G. Trabuco, and K. Schulten. Accelerating molecular modeling applications with graphics processors. *Journal of computational chemistry*, 28(16):2618–2640, 2007.

[30] C. Sun, D. Lo, S.-C. Khoo, and J. Jiang. Towards more accurate retrieval of duplicate bug reports. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 253–262. IEEE Computer Society, 2011.

[31] C. Sun, D. Lo, X. Wang, J. Jiang, and S.-C. Khoo. A discriminative model approach for accurate duplicate bug report retrieval. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 45–54. ACM, 2010.

[32] A. Sureka and P. Jalote. Detecting duplicate bug report using character n-gram-based features. In *2010 Asia Pacific Software Engineering Conference*, pages 366–374. IEEE, 2010.

[33] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun. An approach to detecting duplicate bug reports using natural language and execution information. In *Proceedings of the 30th international conference on Software engineering*, pages 461–470. ACM, 2008.

[34] P. Wittek and S. DaráNyi. Accelerating text mining workloads in a mapreduce-based distributed gpu environment. *J. Parallel Distrib. Comput.*, 73(2):198–206, 2013.

[35] Y. Zhang, F. Mueller, X. Cui, and T. Potok. Gpu-accelerated text mining. In *Workshop on exploiting parallelism using GPUs and other hardware-assisted methods*, 2009.