UNIVERSIDADE FEDERAL DE GOIÁS
INSTITUTO DE INFORMÁTICA

LEONARDO AFONSO AMORIM

Exploiting Parallelism in Document Similarity Tasks with Application

Goiânia, Brazil
2019

TERMO DE CIÊNCIA E DE AUTORIZAÇÃO PARA DISPONIBILIZAR VERSÕES ELETRÔNICAS DE TESES E DISSERTAÇÕES NA BIBLIOTECA DIGITAL DA UFG

Na qualidade de titular dos direitos de autor, autorizo a Universidade Federal de Goiás (UFG) a disponibilizar, gratuitamente, por meio da Biblioteca Digital de Teses e Dissertações (BDTD/UFG), regulamentada pela Resolução CEPEC nº 832/2007, sem ressarcimento dos direitos autorais, de acordo com a Lei nº 9610/98, o documento conforme permissões assinaladas abaixo, para fins de leitura, impressão e/ou *download*, a título de divulgação da produção científica brasileira, a partir desta data.

1. Identificação do material bibliográfico: **[ ] Dissertação** **[X] Tese**

2. Identificação da Tese ou Dissertação:

Nome completo do autor: Leonardo Afonso Amorim

Título do trabalho: Exploiting Parallelism in Document Similarity Tasks with Application

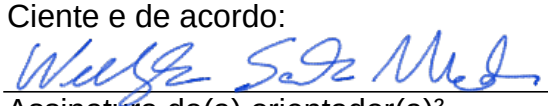
3. Informações de acesso ao documento:

Concorda com a liberação total do documento [X] SIM [ ] NÃO[1]

Havendo concordância com a disponibilização eletrônica, torna-se imprescindível o envio do(s) arquivo(s) em formato digital PDF da tese ou dissertação.

Assinatura do(a) autor(a)[2]

Ciente e de acordo:



Assinatura do(a) orientador(a)[2] Data: 04/10/2019

[1] Neste caso o documento será embargado por até um ano a partir da data de defesa. A extensão deste prazo suscita justificativa junto à coordenação do curso. Os dados do documento não serão disponibilizados durante o período de embargo.

Casos de embargo:

- Solicitação de registro de patente;
- Submissão de artigo em revista científica;
- Publicação como capítulo de livro;
- Publicação da dissertação/tese em livro.

[2] A assinatura deve ser escaneada.

LEONARDO AFONSO AMORIM

Exploiting Parallelism in Document Similarity Tasks with Application

Thesis presented to the postgraduate program of Instituto de Informatica from Universidade Federal de Goias, as a partial fulfillment of the requirements for the Ph.D. degree in Computer Science.

Área de concentração: Ciência da Computação.

Orientador: Prof. Dr. Wellington Santos Martins

Goiânia, Brazil
2019

Ficha de identificação da obra elaborada pelo autor, através do
Programa de Geração Automática do Sistema de Bibliotecas da UFG.

Afonso Amorim, Leonardo
Exploiting Parallelism in Document Similarity Tasks with Application [manuscrito] / Leonardo Afonso Amorim. - 2019.
111 f.: il.

Orientador: Prof. Dr. Wellington Santos Martins.
Tese (Doutorado) - Universidade Federal de Goiás, Instituto de Informática (INF), Programa de Pós-Graduação em Ciência da Computação em rede (UFG/UFMS), Goiânia, 2019.
Bibliografia.
Inclui tabelas, algoritmos, lista de figuras, lista de tabelas.

1. knn. 2. textual datasets. 3. fine-grained parallel algorithm. 4. Word embedding. 5. Document Similarity Tasks. I. Santos Martins, Wellington, orient. II. Título.

CDU 004

UNIVERSIDADE FEDERAL DE GOIÁS

INSTITUTO DE INFORMÁTICA

ATA DE DEFESA DE TESE

Ata Nº **03/2019** da sessão de Defesa de Tese de Leonardo Afonso Amorim que confere o título de Doutor em Ciência da Computação, na área de concentração em Ciência da Computação.

Aos cinco dias do mês de setembro de dois mil e dezenove a partir das oito horas e trinta minutos, na sala 150 do Instituto de Informática, realizou-se a sessão pública de Defesa de Tese intitulada **"Exploiting Parallelism in Document Similarity Tasks with Application**". Os trabalhos foram instalados pelo Orientador, Professor Doutor Wellington Santos Martins (INF/UFG) com a participação dos demais membros da Banca Examinadora: Professor Doutor Auri Marcelo Rizzo Vincenzi (DC/UFSCar), membro titular externo, cuja participação ocorreu através de videoconferência; Professor Doutor Weber Martins (EMC/UFG), membro titular externo; Professor Doutor Cássio Leonardo Rodrigues (INF/UFG), membro titular interno; Professor Doutor Thierson Couto Rosa (INF/UFG), membro titular interno. Durante a argüição os membros da banca não fizeram sugestão de alteração do título do trabalho. A Banca Examinadora reuniu-se em sessão secreta a fim de concluir o julgamento da Tese tendo sido o candidato **aprovado** pelos seus membros. Proclamados os resultados pelo Professor Doutor Wellington Santos Martins, Presidente da Banca Examinadora, foram encerrados os trabalhos e, para constar, lavrou-se a presente ata que é assinada pelos Membros da Banca Examinadora, aos cinco dias do mês de setembro de dois mil e dezenove.

TÍTULO SUGERIDO PELA BANCA

Documento assinado eletronicamente por **Wellington Santos Martins**, **Professor do Magistério Superior**, em 05/09/2019, às 11:28, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Documento assinado eletronicamente por **Thierson Couto Rosa**, **Professor do Magistério Superior**, em 05/09/2019, às 11:28, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Documento assinado eletronicamente por **Cássio Leonardo Rodrigues**, **Coordenador de Curso**, em 05/09/2019, às 11:28, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Documento assinado eletronicamente por **Weber Martins**, **Professor do Magistério Superior**, em 05/09/2019, às 11:29, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Documento assinado eletronicamente por **Auri Marcelo Rizzo Vincenzi**, **Usuário Externo**, em 05/09/2019, às 13:11, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

A autenticidade deste documento pode ser conferida no site https://sei.ufg.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **0816227** e o código CRC **2B3BFB87**.

Referência: Processo nº 23070.024896/2019-11 SEI nº 0816227

All rights reserved. It is forbidden the total or partial reproduction of the work without permission from the university, author and advisor.

Leonardo Afonso Amorim (bibliograhic citation: AMORIM, L. A.)

Researcher of Machine Learning Algorithms and Implementations with High Performance and Big Data. Solid knowledge in Linux infrastructure, Big Data with Hadoop Ecosystem, Java / Python programming, and teaching. Experience obtained in masters and doctorates with the following algorithms: kNN, kMeans, Association Rules, Artificial Neural Networks, Natural Language Processing with Word2Vec, Multiagent Systems, Genetic Algorithms, and Genetic Programming. He works with Machine Learning application for Automated Software Repair. He has lectured on Artificial / Computational Intelligence for Computer Engineering at the Federal University of Goias. He is a member of the research group of High Performance and Applications (http://dgp.cnpq.br/dgp/espelhogrupo/7985061476854055) and I develop parallel and scalable algorithms for document search tasks by similarity (research involves algorithms like KNN and Word2Vec) using fine granularity processing (GPGPUs).He is a member of the research group Intelligence for Software (http://ic1.inf.ufg.br/i4soft/team.html) and I develop new methods and algorithms for Automated Software Repair using Machine Learning (especially Artificial Neural Networks and Word Embeddings). He researches efficient and scalable algorithms and their parallel implementations in multicore and manycore architectures to speed up document search tasks by similarity using Machine Learning techniques. He received a CAPES scholarship in 2015, which has contributed to the development of this Ph.D. thesis.

This PhD thesis is specially dedicated to God, my parents, my son and my wife.

Agradecimentos

À minha mãe Celuta Maria Afonso que trabalhou sem descanso para me proporcionar tudo que tenho hoje e por ter sido meu exemplo de amor, conduta e sabedoria. Ao meu pai Jair Jerônimo Sobrinho (em memória) ao me dizer, com voz embargada antes de partir para a eternidade, que eu seria um doutor. À minha esposa Paloma Victor Santos Amorim pelo amor, paciência e apoio constante. À minha sogra Isabel Moreira Victor pelo apoio dado para a minha família, em especial, ao meu filho durante essa fase do doutorado. Ao meu filho Heitor Leonardo Victor Amorim por me dar forças com seu sorriso e alegria nos momentos mais desafiadores, embora, neste momento, ele tenha apenas dois anos de idade. Ao meu irmão Jairo Afonso Amorim (em memória). Especialmente ao meu irmão Jader Afonso Amorim por ter me dado suporte financeiro desde o início da minha vida estudantil e pelo apoio moral durante minhas dificuldades. Às minhas irmãs Célia Maria Amorim, Maria Isabel Amorim, Rosimeire Amorim e Sandra Regina Amorim pelo apoio em todos os momentos de minha vida.

Gostaria de agradecer também aos excelentes colegas de pós graduação Mateus Ferreira Freitas, Paulo Henrique Silva, Altino Dantas e Eduardo Souza por terem colaborado nos experimentos e discussões deste trabalho. Aos colegas de grupo de pesquisa Evandro Carrijo Taquary, Savio Salvarino Teles de Oliveira e Roussian Gaioso. E também ao colega de doutorado Lauro Cássio Martins de Paula.

Aos meus orientadores, professores de mestrado e graduação que me inseriram no mundo da pesquisa acadêmica: Vinicius Sebba Patto, Renato Bulcão, Iwens Gervasio Sene Junior. Ao meu orientador Professor Wellington Santos Martins, minha referência como pesquisador, sempre me instigando com perguntas que me nortearam bastante na minha pesquisa. Aos Professores Celso Gonçalves Camilo Junior e Weber Martins pela atenção e apoios constantes durante esta etapa da minha vida acadêmica.

Superar cada obstáculo na busca dos meus objetivos foi menos árduo com o apoio de cada uma destas pessoas. Chegar até aqui é uma grande vitória que tem valiosa contribuição de cada uma delas e, portanto, é plural.

A CAPES pelo apoio financeiro, indispensável para a realização deste trabalho.

E, principalmente, a Deus, por meio de Cristo Jesus, por ter me permitido alcançar este objetivo me dando forças, fé e colocando as pessoas fundamentais ao meu

redor para o meu crescimento pessoal e intelectual. A Deus toda honra, toda glória, todo louvor e toda gratidão.

“You shall know a word by the company it keeps.”

J. R. Firth,

.

Abstract

Amorim, Leonardo Afonso. **Exploiting Parallelism in Document Similarity Tasks with Application**. Goiânia, Brazil, 2019. 107p. PhD. Thesis . Instituto de Informática, Universidade Federal de Goiás.

The amount of data available continues to grow rapidly and much of it corresponds to text expressing human language, that is unstructured in nature. One way of giving some structure to this data is by converting the documents to a vector of features corresponding to word frequencies (term count, tf-idf, etc) or word embeddings. This transformation allows us to process textual data with operations such as similarity measure, similarity search, classification, among others. However, this is only possible thanks to more sophisticated algorithms which demand higher computational power. In this work, we exploit parallelism to enable the use of parallel algorithms to document similarity tasks and apply some of the results to an important application in software engineering. The similarity search for textual data is commonly performed through a k nearest neighbor search in which pairs of document vectors are compared and the k most similar are returned. For this task we present FaSST-kNN, a fine-grain parallel algorithm, that applies filtering techniques based on the most common important terms of the query document using tf-idf. The algorithm implemented on a GPU improved the top k nearest neighbors search by up to 60x compared to a baseline, also running on a GPU. Document similarity using tf-idf is based on a scoring scheme for words that reflects how important a word is to a document in a collection. Recently a more sophisticated similarity measure, called word embedding, has become popular. It creates a vector for each word that indicates co-occurrence relationships between words in a given context, capturing complex semantic relationships between words. In order to generate word embeddings efficiently, we propose a scalable fine-grain parallel algorithm that creates the embeddings. The algorithm implemented on a multi-GPU system scaled linearly and was able to generate embeddings 13x faster than the original multicore Word2Vec algorithm while keeping the accuracy of the results at the same level as those produced by standard word embedding programs. Finally, we applied our accelerated word embeddings solution to the problem of assessing the quality of fixes in Automated Software Repair. The proposed implementation was able to deal with large corpus, in a computationally efficient way, being a promising alternative to the processing of million source code files needed for this task.

Keywords

Parallel Computing, Document Similarity Tasks, *k*NN, TOP-*k*, Word Embeddings, Word2Vec, Automatic Program Repair

Resumo

Amorim, Leonardo Afonso. **Exploiting Parallelism in Document Similarity Tasks with Application**. Goiânia, Brazil, 2019. 107p. Tese de Doutorado . Instituto de Informática, Universidade Federal de Goiás.

A quantidade de dados disponíveis continua a crescer rapidamente e muitos desses dados correspondem a textos que expressam a linguagem humana, que é de natureza não estruturada. Uma maneira de dar alguma estrutura a esses dados é convertendo os documentos em um vetor de recursos correspondentes a frequências de palavras (contagem de termos, tf-idf etc.) ou construindo palavras com representação latente. Essa transformação nos permite processar dados textuais com operações como medida de similaridade, pesquisa de similaridade, classificação, entre outras. No entanto, isso só é possível graças a algoritmos mais sofisticados que exigem maior poder computacional. Neste trabalho, exploramos o paralelismo para permitir o uso de algoritmos eficientes para realizar tarefas de busca de documentos por similaridade e aplicar alguns dos resultados a uma importante aplicação em engenharia de software. A busca por similaridade de dados textuais é comumente realizada por meio do algoritmo kNN, os k vizinhos mais próximos, no qual pares de vetores de documentos são comparados e os k mais similares são retornados. Para esta tarefa apresentamos o FaSST-kNN, um algoritmo paralelo de granularidade fina, que aplica técnicas de filtragem baseadas nos termos mais importantes do documento de consulta usando tf-idf. O algoritmo implementado em uma GPU melhorou os k vizinhos mais próximos na busca por até 60x em comparação com uma linha de base, também em execução em uma GPU. A similaridade de documento usando tf-idf é baseada em um esquema de frequência para palavras que reflete a importância de uma palavra para um documento em uma coleção. Recentemente, uma medida de similaridade mais sofisticada, chamada representação latente de palavras (word embedding), se tornou popular. Um vetor é criado para cada palavra. O vetor armazena de forma latente relações de coocorrência entre palavras em um determinado contexto, capturando relações semânticas complexas entre palavras. A fim de gerar eficientemente representação latente de palavras, propomos um algoritmo paralelo e escalável de granularidade fina. O algoritmo implementado em um sistema multi-GPU escala linearmente e foi capaz de gerar representações de palavras 13x mais rápidos que o algoritmo Word2Vec multicore original, mantendo a precisão dos resultados no mesmo nível daqueles produzidos pelo programa padrão de geração de representação de palavras latentes. Finalmente, aplicamos nossa solução acelerada de de representação de palavras latentes ao problema de avaliar a qualidade das correções em Reparo Automatizado de Software. A implementação proposta foi capaz de lidar com grande corpus, de forma computacionalmente eficiente, sendo uma alternativa promissora ao processamento de milhões de arquivos de código fonte necessários para esta tarefa.

Palavras–chave

Computação Paralela, Tarefas de Busca de Documentos por Similaridade, TOP-k, Word Embeddings, Word2Vec, Reparo Automatizado de Software

Contents

List of Figures **16**

List of Tables **18**

List of Algorithms **19**

1 Introduction **20**

2 Parallel Computing **24**
- 2.1 Multicore architectures 25
- 2.2 Manycore architectures 26
- 2.3 Types of parallelism 27
  - 2.3.1 Data parallelism and task parallelism 27
  - 2.3.2 Granularity 28
- 2.4 Taxonomy of computers 29
- 2.5 The Bulk Synchronous Parallel model 29
- 2.6 Summary 32

3 Text Representation and Document Similarity Search **33**
- 3.1 Vector Space Model 33
  - 3.1.1 The bag-of-words model (BoW) 33
  - 3.1.2 Limitations of Bag-of-Words 34
  - 3.1.3 Software that implements the vector space model 35
- 3.2 Document Similarity Search 35
- 3.3 Word/Document Embedding 37
  - 3.3.1 Word2Vec Softmax output layer 39
- 3.4 Summary 45

4 Parallel approaches to Document Similarity Search and TOP-K Applications **46**
- 4.1 k-NN 46
- 4.2 TOP-k Applications 47
- 4.3 Related works 48
  - 4.3.1 GT-kNN - the base algorithm of FaSST-kNN 48
    - Creating the Inverted Index 48
    - Calculating the distances 50
    - Finding the k Nearest Neighbors 52
  - 4.3.2 Other related algorithms 53
- 4.4 Proposed work 54
- 4.5 A Parallel kNN Proposal 55

4.5.1 Data Indexing 55
4.5.2 *k* Nearest Neighbors Search 56
4.5.3 Threshold-based Filtering 57
Sampling Method #1 57
Sampling Method #2 57
4.5.4 Fast Similarity Search for Text (FaSST-*k*NN) 58
4.5.5 Multi-Query *k*NN Search 63
4.5.6 Scalable Fast Similarity Search for Text (SFaSST-*k*NN) 64
4.6 Summary 65

5 Parallel approaches to accelerate word embedding generation with fine-grain parallelism 67
5.1 Related works 67
5.2 Parallel solution 69
5.3 Application 71
5.3.1 Approach 73
5.3.2 Related Work 74
5.4 Summary 76

6 Experiments and Results 77
6.1 FaSST-*k*NN 77
6.1.1 Experimental Evaluation 77
Computational Time 78
Runtime Profiling 79
6.2 SFaSST-*k*NN 80
6.2.1 Experimental Evaluation 80
Computational Time 81
Runtime Profiling 84
6.3 Accelerating word embedding generation with fine-grain parallelism 86
6.3.1 Experimental Evaluation 86
Corpus and Parameterization 86
Defects4J Preprocessing 87
Experiments Discussion 87
6.3.2 Results 88
6.4 A new word embedding approach to evaluate potential fixes for APR 88
6.4.1 Experiments 89
Setup 89
Metrics 89
Scenarios 90
6.4.2 Results 90
6.5 Summary 94

7 Conclusions 95
7.1 Summary of contributions 96
7.1.1 Discussion and Limitations of our proposals 97
7.1.2 Published papers 97
7.1.3 Manuscripts in review process 98
7.1.4 Main award 98

7.1.5 Future works 98

References **100**

List of Figures

2.1 A typical chip multithreaded, multi-core system 26
2.2 AGPU Model [38] 27
2.3 Networking communication in BSP model 30
2.4 Supersteps 31

3.1 Cosine similarity 37
3.2 CBOW model with a one-word context setting [65] 42
3.3 CBOW model with a multi-word context setting [65] 43
3.4 Skip-gram model [65] 44

4.1 *Creating the inverted index* 49
4.2 *Example of the execution of Algorithm 4.2 for a query with three terms.* 52
4.3 FaSST-*k*NN flowchart. 58
4.4 Documents. 59
4.5 Inverted Index and Query. 59
4.6 Inverted Index and Query sorted by TF-IDF in descending order. 59
4.7 Sampling method #1 with 6 samples. 60
4.8 Distance calculation. Dot product with term 2. 60
4.9 Distance calculation. Dot product with term 3. 61
4.10 Distance calculation. Dot product with term 1. 61
4.11 Distance calculation. Dividing the dot product by the product of L2-norms. 61
4.12 Example of compaction phase with k=3 and samples=6 62
4.13 Example 2 of compaction phase with k=3 and samples=6 63
4.14 Example of compaction phase with k=2, samples=4 and two distributed indexes. 64

5.1 The proposal workflow. 71
5.2 Proposal's flowchart 73

6.1 Results after Delete operator: (a) Prob (b) Dist 91
91 ((a))
91 ((a))
91 ((b))
91 ((b))
6.2 Results after Insert operator: (a) Prob (b) Dist 92
92 ((a))
92 ((a))
92 ((b))
92 ((b))

6.3 Prob results after Swap operator 92
6.4 Examples of two canditate variants obtained from GenProg and one correct code for smallest problem. 94

List of Tables

6.1 Query times in seconds and speedups to find the K nearest neighbors with 1 GPU. 79
6.2 Query times in seconds and speedups to find the K nearest neighbors with 2 and 4 GPUs. 79
6.3 Impact of the sorted inverted index on the compaction ratio. 80
6.4 Sum of sampling times in seconds. 80
6.5 Sum of sorting times in seconds. 80
6.6 General information on the datasets. 81
6.7 Query times in seconds to find the K nearest neighbors in Medline. 82
6.8 Speedups with 4 GPUs and over NoFilter solutions in Medline. 82
6.9 Query times in seconds and speedups to find the K nearest neighbors in PubMed. 83
6.10 G-KNN comparison with FaSST / SFaSST using 1 GPU and batch size 10. 83
6.11 Compaction ratios for FaSST and SFaSST using 1 and 4 GPUs. 85
6.12 Sum of sampling and compaction times in seconds for Medline with FaSST and SFaSST using 1 and 4 GPUs. 85
6.13 Sum of sorting times in seconds for Medline for all implementations using 1 and 4 GPUs. 85
6.14 Sum of sorting times in seconds for Pubmed for all implementations using 1 and 4 GPUs. 85
6.15 Sum of sampling and compaction times in seconds for Pubmed with FaSST and SFaSST using 1 and 4 GPUs. 86
6.16 Execution time for hidden layer size 200 and 400. 88
6.17 Accuracy percentage for 1Billion with hidden layer size 200 and 400. 88
6.18 Speedup for 4 GPUs x Mikolov 88
6.19 Scores of different variants for each program generated by delete operator level 1 93

List of Algorithms

4.1 $CreateInvetedIndex(E)$ 49
4.2 $DistanceCalculation(invertedIndex, q)$ 51

CHAPTER 1

Introduction

The amount of data available continues to grow rapidly and much of it corresponds to text expressing human language, that is unstructured. One way of giving some structure to this data is by converting the documents to a vector of features corresponding to word frequencies (term count, tf-idf[1], etc) or word/document embeddings[2]. This transformation allows us to process textual data with operations such as similarity measure, similarity search, classification, among others. This transformation implies vectors of high dimensionality[3] and sparsity, which tend to be equidistant in this vector space, a phenomenon known as the curse of dimensionality. This makes it difficult to use traditional algorithms based on distance measures because they are computationally costly in terms of processing and memory usage.

The curse of dimensionality usually refers to what happens when we add more and more variables to a multivariate model. The more dimensions we add to a data set, the more difficult it becomes to process it. It means that the number of dimensions is staggeringly high - so high that calculations become extremely difficult. With high dimensional data, the number of features can exceed the number of observations. For example, textual datasets, which besides having many documents, are commonly related to a vocabulary with hundreds of thousands of words. The statistical curse of dimensionality points to a related fact: a demanded sample size n will increase exponentially with data that has d dimensions. In simple terms, adding more dimensions could mean that the sample size we need suddenly become uncontrollable. Thus, more complex and computationally expensive algorithms are required.

High-dimensional datasets arise in diverse areas ranging from computational advertising to natural language processing. Learning in such high-dimensions can be limited in terms of computations and memory. To this purpose, techniques for dimensionality re-

[1] In information retrieval, tf-idf or TFIDF, short for term frequency-inverse document frequency, is a numerical statistic that is intended to reflect how important a word is to a document in a corpus [50].

[2] Before generating a word embedding, a word is represented by a high-dimensional vector of vocabulary size in one-hot encoding format.

[3] Dimensionality in statistics regards to how many attributes a dataset has.

duction like (linear/nonlinear) principal component analysis or manifold learning are used for transformation in a lower-dimensional space. However, these dimensionality reduction methods, seldom, cannot be applicable, e.g., in sparse datasets that have independent features and the data lie in multiple lower-dimensional manifolds.

An example of an algorithm used to process high dimensionality and sparse datasets is the k nearest neighbors (kNN). Finding the closest k elements in metric spaces is an optimization problem. Many techniques have used to tackle this problem. One approach to solving the kNN problem is an exhaustive search technique (also known as brute-force) to find the nearest neighbors of a point. In this technique, all distances between a query q and the points in metric space are computed. The next step is sorting the computed distances and selecting the k elements corresponding to the k smallest distances. The disadvantage of this approach is the high computational cost: $\mathcal{O}(nd)$ for calculating the n distances and $\mathcal{O}(nlogn)$ for sorting them.

In this context, the computational power of today's processors can help us to deal with processing high dimensionality and sparsity textual datasets [1]. It is based on the concept of parallel processing, where multiple cores (multicore / manycore - CPU / GPU) operate concurrently. These parallel computers can provide high performance, but present a major challenge for their programming, as they require the development of new (parallel) algorithms. However, this has been the solution for automating the information extraction process from the huge unstructured Big Data available today. High-Performance Computing (HPC), coupled with data mining techniques, is turning data storage, manipulation, and analysis into tasks cheaper and faster [1], [11]. This has allowed Information Retrieval on unstructured data quickly, such as search engines (eg Google). In addition to searching, the use of HPC has made Data Mining, that is, finding patterns and hidden connections between data more efficient. Finally, HPC has contributed to the refinement of Machine Learning techniques, allowing the generalization of existing knowledge with new data. In this context, there are two open questions which are going to guide this research:

- How to exploit parallelism in document similarity tasks in an efficient and scalable way?
- How to apply the parallel proposals in the context of Automated Program Repair [4] and Top-k applications?

In this research, we exploit parallelism to enable the use of parallel algorithms to document similarity tasks and apply some of the results to an important application

[4] Automated Program Repair is a fast-growing research area in computer science. Some numerous tools and techniques have been developed to automatically find and fix bugs in source code. This topic is directly related to Quality Engineering and improving the quality of software by reducing the cost and effort of fixing bugs [43].

in software engineering. The similarity search for textual data is commonly performed through a k nearest neighbor search in which pairs of document vectors are compared and the k most similar are returned. For this task, we present SFaSST-kNN, a fine-grained parallel algorithm, that applies filtering techniques based on the most common important terms of the query document using tf-idf, and uses a distributed inverted index to perform batch processing of multiple queries. The algorithm implemented on a GPU improved the top k nearest neighbors search by up to 60x compared to a baseline, also running on a GPU.

Document similarity using tf-idf is based on a scoring scheme for words that reflects how important a word is to a document in a collection [50]. Recently a more complex algorithm, called Word2Vec, to generate word embedding allows semantic similarity measure between words and documents. It creates a vector for each word that indicates co-occurrence relationships between words in a given context, capturing complex semantic relationships between words. To generate word embeddings efficiently, we propose a fine-grain parallel algorithm that finds a sample of more dissimilar vectors to find accurately which words co-occur. This technique allows using a small fraction of all the vocabulary to distribute the probabilities of a word being similar to one word and all others. The algorithm implemented on a multi-GPU system scaled linearly and was able to generate embeddings 13x faster than the original multicore Word2Vec algorithm while keeping the accuracy of the results at the same level as those produced by standard word embedding programs. Finally, we applied our accelerated word embeddings solution to the problem of assessing the quality of fixes in Automated Software Repair. The proposed implementation was able to deal with large corpus[5], in a computationally efficient way, being a promising alternative to the processing of million source code files needed for this task.

The main contributions of this research are: (i) A threshold-based filtering technique that improves the sorting time of kNN candidates; (ii) A scalable multi-GPU implementation that exploits both data parallelism and task parallelism; (iii) A distributed inverted index implementation in MultiGPU system; (iv) The possibility of batch processing multiple queries; (v) Extensive experimental work with a standard real-world textual datasets; (vi) A multi-GPU implementation that generates word embeddings and achieves linear speed-up on a multi-GPU machine while maintaining the accuracy of the results; (vii) Extensive experimental evaluation of the proposed implementation in both texts from natural language and source codes from programming languages.

We organized this work as follows. Section 2 provides background to parallel computing concepts. Section 3 provides background to text representation and document

[5]Millions or billions of words.

similarity search. Section 4 presents a parallel approach to Document Similarity Search and Top-k Applications. Section 5 presents a parallel approach to accelarate word embedding generation with fine-grain parallelism. Section 6 presents our experiments and results. Section 7 presents our conclusions.

CHAPTER 2

Parallel Computing

The computational power of today's processors can help us to deal with processing high dimensionality and sparsity textual datasets. It is based on the concept of parallel processing, where multiple cores (multicore / manycore - CPU / GPU) operate concurrently. These parallel computers can provide high performance, but present a major challenge for their programming, as they require the development of new (parallel) algorithms. However, this has been the solution for automating the information extraction process from the huge unstructured Big Data available today. High-Performance Computing (HPC), coupled with data mining techniques, is turning data storage, manipulation, and analysis into tasks cheaper and faster. This has allowed Information Retrieval on unstructured data quickly, such as search engines (eg Google).

For more than 40 years, sequential computer architectures and algorithms have dominated computing systems. However, multicore and manycore architectures have become attractive devices for current computing challenges. As time went on, Parallel Computing has established itself in the market and, as a consequence, in the development of algorithms. Faced with this, the algorithm designers have been confronted with several challenges that previously did not exist.

Parallel algorithms must be competitive compared to sequential algorithms. Sequential algorithms have advantages like simplicity and are often highly optimized. A constant factor (usually the number of processors) commonly limits the speedup of a parallel algorithm. Currently, there are different forms of parallel architectures and, therefore, the speedup on one machine may not be the same on other computers. Therefore, it is necessary for a parallel programming model to abstract the details of the actual parallel machines. This is important since the utilization of parallel computing resources can be underused. A good parallel model allows achieving portability for the parallel algorithm on different parallel platforms. When scaling to a larger number of processors, data and work distribution is also an issue and more sophisticated load distribution models need to be investigated. This section aims to expose some of the parallel computing models and their practical realizations in the form of multicore and manycore architectures.

2.1 Multicore architectures

Two important factors have made unsustainable the continuous increase in performance of sequential processor architectures: increasing energy consumption and high temperature. These factors has driven the industry to create new solutions. The solution has been to integrate multiple cores with low frequency and reduced power consumption on a single chip. As a result, continuous improvement in processor performance has become feasible. Two approaches stood out in this new solution: multicore and manycore architectures. The multicore architecture gathers two or more cores (currently from 2 up to 18) in a single processor to maintain more tasks executing simultaneously. In other words, a chip with more than one CPU's(Central Processing Unit). Some of the most used multicore architectures are AMD Phenom and Ryzen, ARM, Intel Atom, Intel Core i3, Core i5, Core i7 and Core i9 etc.

A multicore processor is a computer processor with two or more separate processing units, called cores. Multicore processors allow more efficient simultaneous processing of multiple tasks, making them more powerful than single-core processors. At the same clock frequency, the multicore processor can process more data than the single core processor. In addition, multicore processors can deliver high performance and handle complex tasks at comparatively lower energy compared with a single core.

The system represented in Figure 2.1 has two processors, each with two cores and each core has two hardware threads. There are one L1[1] cache and one L2 cache per core. Each core can have its L2 cache or the cores share the L2 cache. All cores and processors share the system bus to access the main memory or RAM.

[1] A cache is a smaller, faster memory, closer to a processor core, which stores copies of the data from frequently used main memory locations. Most CPUs have different independent caches, including instruction and data caches, where the data cache is usually organized as a hierarchy of more cache levels (L1, L2, L3, L4)

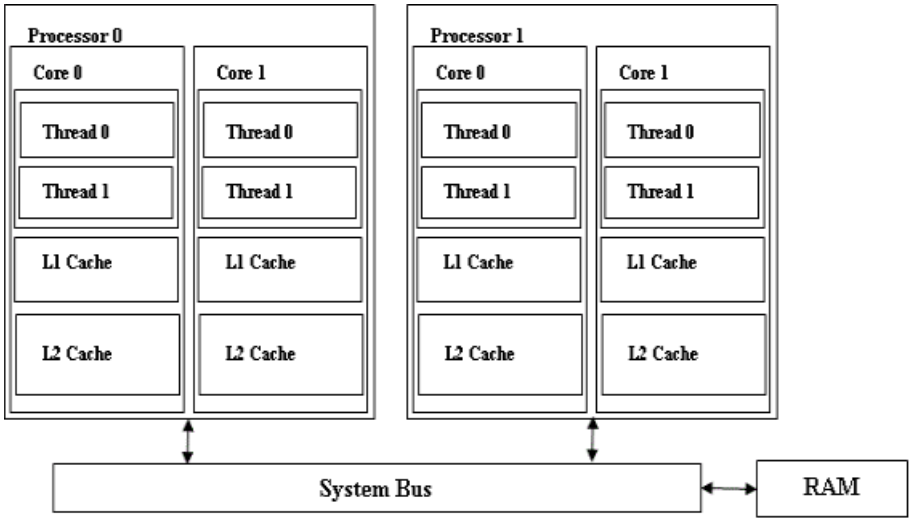


Figure 2.1: *A typical chip multithreaded, multi-core system*

2.2 Manycore architectures

The manycore architecture uses a large number of simpler, independent processor cores to accelerate parallel programs. Manycore processors are distinct from multicore processors in that they are optimized for a higher degree of explicit parallelism, and for higher throughput (or lower power consumption) at the expense of latency and lower single thread performance. GPUs, which can be described as manycore vector processors[2], may be considered a form of manycore processor. Intel Xeon Phi also represents a manycore architecture.

Uniting multiple cores on a single chip provided intense changes in the technology community, especially in the parallel algorithm development paradigm. As a consequence, the optimal performance of the parallel algorithm often requires in-depth knowledge of the parallel architecture used. NVIDIA has surpassed in the field of machine learning, especially in deep learning by providing CUDA Toolkit[3] (constantly updated) with improvements to the memory model, profiling tools, and new libraries for these areas.

Recently, hardware manufacturers propose many GPU architectures with similar characteristics. However, we will focus on the NVIDIA design, which stands out with its CUDA architecture. The GPU environment consists of a *host* (CPU) and a *device* (GPU). The device consists of p cores and one *global memory unit*. Each core handles a single thread and executes one instruction per unit time. The word length of the device is

[2] In computing, a vector processor or array processor is a central processing unit (CPU) that performs an instruction set containing instructions that operate on one-dimensional arrays of data called vectors, compared to scalar processors, whose instructions operate on single data items.

[3] https://developer.nvidia.com/cuda-toolkit

w bits. A group of b cores forms a *multiprocessor*. The device has k multiprocessors, that is, $p = kb$. Each multiprocessor has its *shared memory unit* with M words and individually executes programs launched by the host [38]. The Figure 2.2 shows a simplified architecture of a GPU.

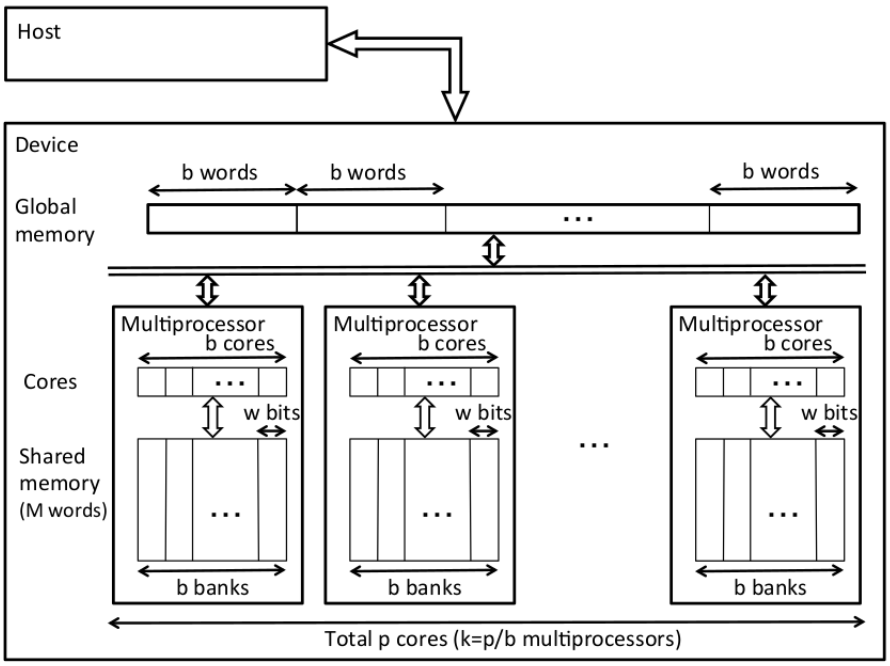


Figure 2.2: *AGPU Model [38]*

Multiprocessors do not communicate with each other. The host can synchronize the multiprocessors by waiting for all multiprocessors in the device to complete executing programs. All cores in a multiprocessor run the same instruction at the same time, but data addresses of their operands can be arbitrary [38].

The global memory unit is high-capacity, low-speed and can be accessed by the host and all multiprocessors in the device, whereas the shared memory units are low-capacity, high-speed and can be accessed only by cores in the multiprocessor. The GPU presents the global memory unit into blocks of b words.

2.3 Types of parallelism

2.3.1 Data parallelism and task parallelism

Data parallelism is parallelization over many processors in parallel computing environments. It focuses on distributing the data across different nodes, which operate on the data in parallel. It can be applied to regular data structures like arrays and matrices by working on each element in parallel.

A variety of data-parallel programming environments are available now, most popularly employed of which are:

- Message Passing Interface is a cross-platform message-passing programming interface for parallel computers. It defines the semantics of library functions to allow users to write portable message-passing programs in C, C++, and Fortran;
- Open Multi-Processing (OpenMP) is an Application Programming Interface (API) which supports shared-memory programming models on multiple platforms of multiprocessor systems;
- CUDA and OpenACC are parallel computing API platforms designed to allow programmers to utilize GPU's computational units for general purpose processing.

Task parallelism, also known as control parallelism, is a form of parallelization of computer code across various processors in parallel computing environments. Task parallelism focuses on distributing tasks -concurrently performed by processes or threads across different processors. In contrast to data parallelism which involves running the same task on different components of data, task parallelism is distinguished by running many different tasks at the same time on the same data.

2.3.2 Granularity

In parallel computing, granularity (or grain size) of a task is a measure of the amount of work (or computation) which is performed by that task [29]. Another definition of granularity takes into account the communication overhead between multiple processors or processing elements. It defines granularity as the ratio of computation time to communication time, wherein, computation time is the time required to perform the computation of a task and communication time is the time required to exchange data between processors [41]. Depending on the amount of work which is performed by a parallel task, parallelism can be classified into couple categories: fine-grained and coarse-grained parallelism.

In fine-grained parallelism, a program is broken down into a large number of small tasks. These tasks are assigned individually to many processors. The amount of work associated with a parallel task is low and the work is evenly distributed among the processors. Hence, fine-grained parallelism facilitates load balancing. As each task processes less data, the number of processors required to perform the complete processing is high. This, in turn, increases communication and synchronization overhead. Fine-grained parallelism is best exploited in architectures which support fast communication. Shared memory architecture which has a low communication overhead is most suitable for fine-grained parallelism.

In coarse-grained parallelism, a program is split into large tasks. Due to this, a large amount of computation takes place in processors. This might result in load imbalance, wherein certain tasks process the bulk of the data while others might be idle. Further, coarse-grained parallelism fails to exploit the parallelism in the program as most of the computation is performed sequentially on a processor. The advantage of this type of parallelism is low communication and synchronization overhead.

Granularity affects the performance of parallel computers. Using fine grains or small tasks results in more parallelism and hence increases the speedup [13]. However, synchronization overhead, scheduling strategies can negatively impact the performance of fine-grained tasks. To reduce communication overhead, granularity can be increased. Coarse-grained tasks have less communication overhead but they often cause load imbalance. Hence the optimal performance is achieved between the two extremes of fine-grained and coarse-grained parallelism [83].

2.4 Taxonomy of computers

Taxonomy of Flynn traditionally has classified computers since 1996. Two factors are essential for this method: instruction flow and data flow. From these two factors it is possible to obtain four categories:

1. **Single instruction, single data stream - SISD**: A single instruction is executed on a single datum during each fetch-execute cycle;
2. **Single instruction, multiple data stream - SIMD**: Single instruction executed on multiple data. This allows us to perform vector or matrix operations with single instructions. Two forms of SIMD architectures are: Vector processors (each processor operates on 1 datum of a 1-D array) and Matrix processors (each processor operations on 1 datum of a 2-D array);
3. **Multiple instruction, single data stream - MISD**: is a type of parallel computing architecture where many functional units perform different operations on the same data. Pipeline architectures belong to this type, though a purist might say that the data is different after processing by each stage in the pipeline.
4. **Multiple instruction, multiple data stream- MIMD**: MIMD are true parallel processors, i.e, each unit is a full processor (control unit, ALU, cache).

2.5 The Bulk Synchronous Parallel model

There is not yet a preferred model adopted by the scientific community. In this chapter, we present the Bulk Synchronous Parallel (BSP) model since it has been

successfully used when mapping a fine-grain parallel algorithm to the GPU.

The BSP model was developed by Leslie Valiant of Harvard University during the 1980s. This model focuses on three important concepts: an abstract parallel machine, a BSP parallel algorithm, and h-relation. A BSP computer consists of:

1. components capable of processing and local memory transactions (i.e., processors);
2. a network that routes messages between pairs of such elements;
3. a hardware facility that allows for the synchronization of all or a subset of components.

Intended to be employed for distributed-memory computing, the model assumes a BSP machine consists of p same processors, which can execute r floating point operations per second. Each processor has access to its local memory. These processors can communicate with each other through an all-to-all network, providing uniform point-to-point access times and bandwidth capacity. The Figure 2.3 shows an abstract machine of the BSP model. Where P is a processor and M a memory.

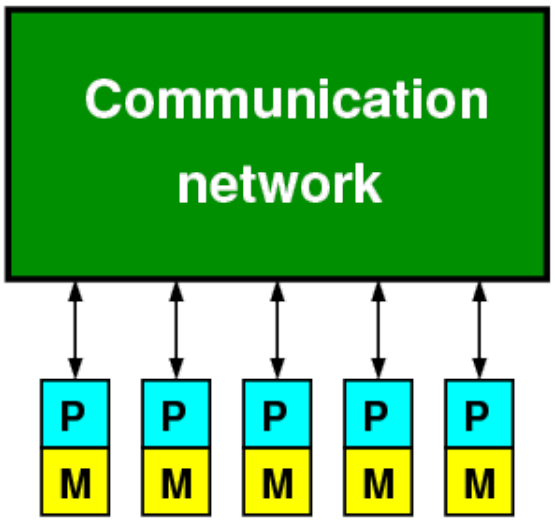


Figure 2.3: *Networking communication in BSP model*

The interconnection network is abstracted by the two-parameter model L and g. The parameter L represents the time to synchronize all processors. The parameter g represents the ability of a communication network to deliver data. These parameters are determined experimentally by each parallel computer. With this method of abstracting parallel architectures, this model approaches real parallel machines [74].

In many parallel programming systems, communications are considered at the level of individual actions: sending and receiving a message, memory to memory transfer, etc. The BSP model considers communication actions *en masse*. In BSP, a parallel program runs across a set of virtual processors (called processes to distinguish them from physical processors) and executes as a sequence of parallel super steps separated by

barrier synchronizations. Each super step is composed of three ordered phases, as shown in Figure 2.4. The Supersteps consist of the following steps:

1. **Concurrent computation**: every participating processor may perform local computations, i.e., each process can only make use of values stored in the local fast memory of the processor. The computations occur asynchronously of all the others but may overlap with communication;
2. **Communication**: The processes exchange data between themselves to facilitate remote data storage capabilities;
3. **Barrier synchronization**: When a process reaches this point (the barrier), it waits until all other they have achieved the same barrier.

In the BSP model, the cost complexity is given by the function: $T(h) = w + hg + L$, where w is the longest computing time, the parameter h represents the maximum number of incoming or outgoing messages for a super steep . Only by going through the three stages of a super step, the algorithm goes to the next one. The cost of the algorithm then is the sum of the costs of each super step.

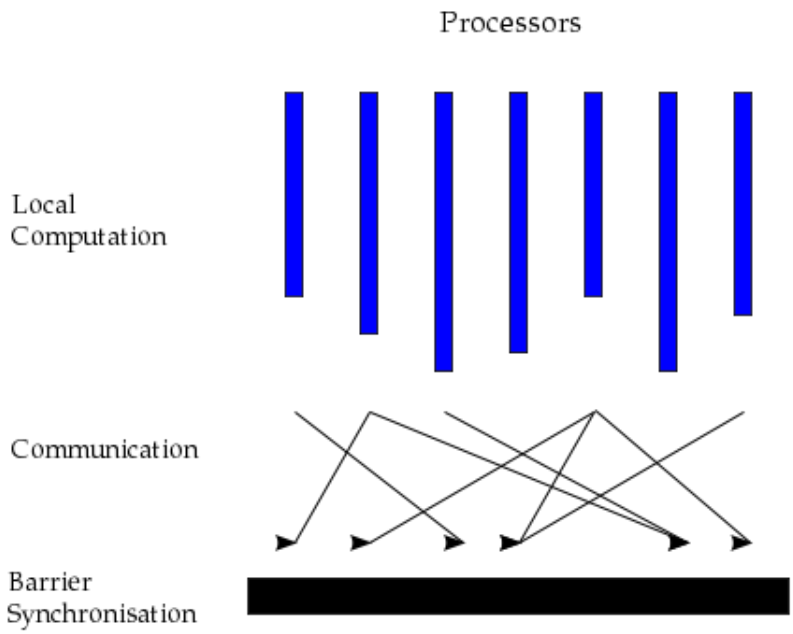


Figure 2.4: *Supersteps*

The BSP model works similarly to the Parallel Random Access Machine (PRAM) model. It can be considered a generalization of the PRAM model, when the BSP machine has the parameter value g small (g = 1) [74]. At a high level, BSP algorithms

can be mapped into super steps, with the bulk synchronization happening in between the kernel launches.

2.6 Summary

In this chapter, we have taken a general approach to the main topics related to parallel computing such as parallel architectures, parallel programming models. In this research, we adopted as a strategy to develop parallel fine-grained data parallelism algorithms for similarity document searching and word embedding generation using Nvidia's CUDA architecture. The CUDA abstractions provide fine-grained data parallelism and task parallelism, nested within coarse-grained data parallelism and task parallelism. They guide the programmer to partition the problem into coarse sub-problems that can be solved independently in parallel by blocks of threads, and each sub-problem into finer pieces that can be solved cooperatively in parallel by all threads within the block. Although most implementations of parallel algorithms were using fine-grained data parallelism, we also used task parallelism to split queries across multiple GPUs into a specific version of the *k*NN algorithm.

CHAPTER 3

Text Representation and Document Similarity Search

The text mining studies are getting more attention recently because of the availability of the increasing number of the electronic documents from a variety of sources. The resources of unstructured and semi structured information include the many large data sets from World Wide Web. Today, the internet is the primary source for text documents and in order to process this data two aspects deserve highlighting: text representation and text comparison. In this research, we focus on two algorithms: k-nearest neighbors (kNN) for text comparison and Word2Vec for text representation.

3.1 Vector Space Model

Vector space model is an algebraic model for representing text documents (and any objects, in general) as vectors of identifiers [67], such as, for example, index terms[1]. It is used in information filtering, information retrieval, indexing. Documents and queries are represented as vectors: $d_i = (w_{1,i}, w_{2,i}, \ldots, w_{t,i})$ and $q = (w_{1,q}, w_{2,q}, \ldots, w_{n,q})$. Each dimension corresponds to a separate term. If a term occurs in the document, its value in the vector is non-zero. Each weight is a feature that measures the importance of an index term in a document or a query, respectively. Several different ways of computing these values, also known as term weights, have been developed.

3.1.1 The bag-of-words model (BoW)

The bag-of-words model (BoW) is a way of extracting features from the text for use in modeling, such as with machine learning algorithms. A bag-of-words is a representation of text that describes the occurrence of words within a document. It means

[1]The definition of the term depends on the application. Typically terms are single words or longer phrases. If words are chosen to be the terms, the dimensionality of the vector is the number of words in the vocabulary - the number of distinct words occurring in the corpus.

couple points: (i) A vocabulary of known words; (ii) a measure of the presence of known words. It is called a "bag" of words because any information about the order or structure of words in the document is discarded. The model is only concerned with whether known words occur in the document, not wherein the document. In other words, BoW is a document representation that computes how many times a word appears in a text. Those word counts allow us to compare texts and calculate their similarities for applications like search, document classification where the frequency of each word expresses a feature for training a classifier. In this model, a document is represented as the bag of its words, ignoring grammar and even word order but keeping multiplicity [50].

Consequently, BoW model is a tool of feature production. After transforming the text into a "bag of words" we can determine many measures to discriminate the document. Terms are document features. Term frequency (TF) is a measure of occasions a word appears in the text. Nevertheless, the most frequent word is a less useful metric because some words like 'this', 'a' occur very frequently across all documents. Consequently, we also need a measure of how unique a word is. Another technique to estimate the topic of a document by the words it contains is Term-frequency-inverse document frequency (TF-IDF)[2]. The IDF weight is defined as $\log(N/n_j)$, i.e., the log of the inverse of the fraction of texts in the whole set which contain term j, where n_j is the number of documents containing term j and the total number of documents is N. Therefore, TF-IDF measures relevance, not frequency, i.e., this metric is a numerical statistic that is intended to reflect how relevant a word is to a text in a corpus.

BoW is distinct from Word2vec, which we will cover next. The principal difference is that Word2vec produces one vector per word or document, whereas BoW provides one number (a word count). Word2vec is useful for texts because it recognizes semantics. The vectors generated by Word2Vec represent each word's context.

3.1.2 Limitations of Bag-of-Words

The bag-of-words model is easy to understand and implement. It has been used on prediction problems like language modeling and document classification. Although, it suffers from some weaknesses, such as:

- Vocabulary: Large vocabulary impacts the sparsity and dimensional of the document representations;
- Sparsity and dimensionality: Sparse and high dimensional representations are harder to model both for computational reasons: space and time complexity;

[2]Tf-IDF is a weight or value that is associated with each document feature.

- Semantic: Discarding word order neglects words context in the document. Context and meaning can offer a lot to the model, that if modeled could tell the difference between the same words differently arranged or synonyms (“old car” versus “used car”), etc.

3.1.3 Software that implements the vector space model

- Apache Lucene: It is a high-performance, full-featured text search engine library written entirely in Java;
- Elasticsearch: It is a high-performance, full-featured text search engine using Lucene;
- Gensim: It is a Python+NumPy framework for Vector Space modelling. It contains incremental (memory-efficient) algorithms for term frequency-inverse document frequency, etc;
- Weka: It is a popular data mining package for Java including Word Vectors and Bag Of Words models;
- Word2vec: It uses vector spaces for word embeddings.

3.2 Document Similarity Search

In recent years, there has been lots of interest in producing effective techniques for non-generalizable search and retrieval in relational databases, document etc. A standard approach to handle this problem is top-k querying. This method provides a ranking from the results and returning the k elements with the highest scores. In the last years, researchers introduced many alternatives to the top-k retrieval problem and several algorithms. In this research, we will cover kNN algorithm. This algorithm can be applied to generate the top-k more similar documents as the result of a query.

k-Nearest Neighbor (kNN) is an algorithm classified as a distance-based method. Generally, the distance measure used in the kNN is the Euclidean or Cosine distance. kNN is an extension of the 1-NN algorithm. Rather than considering only a neighbor, kNN considers k objects in the training set closest to the test point. The term k is a parameter of the algorithm. The choice of k is essential in building the kNN model because it can influence the quality of predictions. This algorithm is one of those algorithms that are very simple to understand and represents one of the most known paradigms of inductive learning: objects with similar characteristics belong to the same group.

kNN assumes that the data is in feature space. This approach allows having a notion of distance between the features. The data used by kNN can be scalars or possibly even multidimensional vectors. This algorithm has k closest training examples in the

feature space as input and the output depends on whether *k*NN is used for classification or regression. In classification, the output is a class membership. In regression, the output is the property value for the object (the average of the values of its *k* nearest neighbors).

The *k*NN algorithm consists of two stages: the training phase and the classification phase. In the training phase, the training examples are vectors in a multidimensional feature space. In this phase, the feature vectors and class labels of training samples are stored. In the classification phase, *k* is a user-defined constant, a query is classified by assigning a label, which is the most recurrent between the *k* training samples nearest to that query point [66].

It is important to note that *k*NN is a non-parametric method used for classification and regression. It considered non-parametric because it does not make any assumptions on the underlying data distribution. It means most of the practical data does not obey the typical theoretical assumptions made. *k*NN is a lazy learning algorithm where its function is approximated locally and all computations are delayed until classification. No actual model is performed during the training phase, although a training dataset is needed, it is used only to populate a sample of the search space with instances whose class is known, for this reason, this algorithm is also known as lazy learning algorithm. It means that the training data points are not used to do any generalization and all the training data is required during the testing phase. When an instance whose class is unknown is presented for evaluation, the algorithm computes its *k* closest neighbors, and the class is assigned by voting among those neighbors. In *k*NN algorithm, training phase is very fast but testing phase is costly in terms of both time and memory.

In other words, the algorithm is considered lazy because all the computation is postponed until the classified phase since the learning process consists only in memorizing the objects. Another important factor is the performance of *k*NN. The training phase requires little computational effort. However, classifying an object (a test sample) requires computing the distance of this object to all training objects. Therefore, prediction can be costly, and for a large set of training objects, this process can be time-consuming. Like all distance-based algorithms, *k*NN is affected by the presence of redundant or irrelevant attributes. Another striking factor in *k*NN performance is related to the dimensionality of the samples. The space defined by the attributes of a problem grows exponentially with the number of attributes. The number of attributes defines the number of dimensions of the space.

Similarity searching algorithm is at the core of document retrieval systems and performance is crucial for it. The *k*NN algorithm ranks the documents closest to a query. For this, it scores queries against documents and computes the highest scoring documents for presentation to the user in ranked order. Representing a document as a vector is the most traditional way of finding the nearest neighbors. In the vector space model [67], a

document is a vector with one component for each unique term (word) in the vocabulary of the corpus. As the vocabulary can be arbitrarily large, the dimensionality of this space is usually very high, with tens of thousands of terms used routinely.

The weight, for that term in a document, represents each word in a text. A function of the frequency of the word, TF-IDF, provides this weight. Representing a document as a vector is also known as a bag-of-words representation. In this model, only the counts of words mattered. In practice, we can use the bag-of-words model as a tool for feature generation. Also, since most coordinates contain terms that do not occur in the document (zero weight), documents are represented by sparse vectors. Vectors also represent queries. Similarity search ranks documents concerning their similarity to the query. The similarity measure most commonly used is the cosine of the angle between the query vector and the document vectors 'Doc 1' and 'Doc 2' as can see in Figure 3.1.

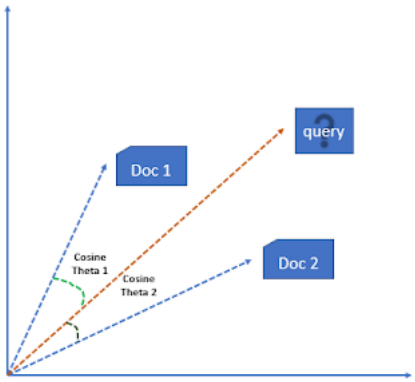


Figure 3.1: *Cosine similarity*

At retrieval time, the documents are ranked by the cosine of the angle between the document vectors and the query vector. For each document and query, the cosine of the angle is calculated as the ratio between the inner product between the document vector and the query vector, and the product of the norm of the document vector by the norm of the query vector. The documents are then returned by the system by decreasing cosine [55].

3.3 Word/Document Embedding

Word embedding is the collective name for a set of language modeling and feature learning techniques in natural language processing where words or phrases from the vocabulary are mapped to vectors of real numbers in a low-dimensional space relative to the vocabulary size ("continuous space"). Methods to generate this mapping include neural networks [42], dimensionality reduction on the word co-occurrence matrix [46], probabilistic models, and explicit representation in terms of the context in which words

appear [45]. Word, phrase and document embeddings, when used as the underlying input representation, have been shown to boost the performance in NLP tasks such as syntactic parsing [75], sentiment analysis [75], automatic document classification [82]. Current methods for generating word embeddings fall into two categories: count-based methods [10] (e.g. Latent Semantic Analysis), and predictive methods (e.g. neural probabilistic language models). In this project, we focus on Word2Vec [56], which is a predictive method with low computational costs that permits the processing of large datasets.

Word2Vec learns continuous word embeddings from plain text in an entirely unsupervised way. The model assumes the Distributional Hypothesis [27], which states that words that appear in the same contexts tend to share semantic meaning. This allows us to estimate the probability of two words occurring close to each other. With Word2Vec, a neural network is trained with streams of n-grams of words so as to predict the n-th word, given words $[1,...,n-1]$ or $[n+1,n+2,...]$. The output is a matrix of word vectors or context vectors. Since the neural network used has a simple linear hidden layer, the intensity of correlation between words is directly measured by the inner product between word embeddings. Such a shallow neural network might not produce as precise distributed representations as deep neural networks on relatively small datasets, but they can process data much more efficiently. When operating on large datasets, this approach usually produces better results than using a more sophisticated algorithm [57].

The two neural network models used by Word2Vec are known as continuous bag-of-words (CBOW) and Skip-gram. Rather than predicting a word conditioned on its predecessor, as in a traditional bi-gram language model, the CBOW model [56] predicts the current word based on the context, while the Skip-gram model [56] predicts the neighborhood words given the current word. The training of these models has to compute normalization terms, which has complexity $\mathcal{O}(|\mathcal{V}|)$, where $|\mathcal{V}|$ is the vocabulary size. To reduce this high computational cost, Word2Vec uses fast training algorithms: hierarchical softmax and negative sampling [46]. Hierarchical softmax makes use of a Huffman tree representation of the vocabulary, which saves calculations, at the potential loss of some accuracy. On the other hand, negative sampling avoids using the words observed next to one another in the training data as positive examples, and instead sample random words from the corpus and present to the network as negative examples. Word2Vec remains a popular choice for building word vectors due to their efficiency and simplicity. Although these fast algorithms are widely used, generating word embeddings is still too costly, which impacts negatively on the time for conducting experiments in both Information Retrieval and Machine Learning applications.

3.3.1 Word2Vec Softmax output layer

In mathematics, the softmax function (normalized exponential function) is a generalization of the logistic function. The softmax function is used to obtain a well-defined probabilistic (multinomial) distribution among words. This function limits the range of the normalized data to values between 0 and 1. These values can be considered as probabilities given an input in the artificial neural network. The softmax function is often used in the final layer of artificial neural networks, which are applied to classification problems.

The weights between the input layer and the output layer can be represented by a $V \times N$ matrix $\mathbf{W}$. Where V is vocabulary size and N is the hidden layer size. Each row of $\mathbf{W}$ is the N-dimensional vector representation $\mathbf{v}_{word}$ of the associated word of the input layer. Given a context word (neighbor word), assuming $x_b = 1$ and $x_{b'} = 0$ for $b' \neq b$, where the b value represents a bit position in a word embedding vector[3], then:

$$\mathbf{h} = \mathbf{x}^T \mathbf{W} := \mathbf{v}_{word_{input}} \tag{3-1}$$

The vector $\mathbf{h}$ represents a word embedding derived from one-hot encoding vector input in Word2Vec Neural Network. In other words, this operation is essentially copying the b-th row of $\mathbf{W}$ to $\mathbf{h}$. $\mathbf{v}_{word_{input}}$ is the vector representation of the input word $word_i$. From the hidden layer to the output layer, there is a different weight matrix $\mathbf{W'} = \text{word'}_{ij}$, which is a $N \times V$ matrix. Using these weights, we can compute a score u_j for each word in the vocabulary:

$$u_j = \mathbf{v'}_{word_j}.\mathbf{h} \tag{3-2}$$

where $\mathbf{v'}_{word_j}$ is the j-th column of the matrix $\mathbf{W'}$. Then we can use softmax to obtain the posterior distribution of words, which is a multinomial distribution [65].

$$p(word_j \mid word_{input}) = y_j = \frac{exp(u_j)}{\sum_{j'=1}^{N} exp(u_{j'})} \tag{3-3}$$

where y_j is the output of the j-th node in the output layer. Substituting (3-1) and (3-2) into (3-3), we obtain equation 3-4:

[3] In other words, only one element of $\{x_1, ..., x_V\}$ is 1, and all other elements are 0.

$$p(word_j \mid word_{input}) = \frac{exp({v'_{word_{output}}}^T v_{word_{input}})}{\sum_{j'=1}^{N} exp({v'_{word'_j}}^T v_{word_{input}})} \tag{3-4}$$

Note that $\mathbf{v}_{word}$ and $\mathbf{v'}_{word}$ are two representations of the word *word*. $\mathbf{v}_{word}$ comes from rows of **W**, which is the input $\rightarrow$ hidden weight matrix, and $\mathbf{v'}_{word}$ comes from columns of **W'**, which is the hidden $\rightarrow$ output matrix. In subsequent analysis, we call $\mathbf{v}_{word}$ as the "input vector", and $\mathbf{v'}_{word}$ as the "output vector" of the word *word*.

The next step is to derive the weight update equation for this model. The training objective is to maximize (3-4), the conditional probability of observing the actual output word *word* (denote its index in the output layer as j^*) given the input context word $word_i$ with regard to the weights.

$$maxp(word_j \mid word_{input}) = u_j^* - log \sum_{j'=1}^{N} exp(u_{j'}) := -E \tag{3-5}$$

where $E = log\ p(word_j \mid word_{input})$ is our loss function (we want to minimize E), and j^* is the index of the actual output word in the output layer. The next step is to derive the update equation of the weights between hidden and output layers. Take the derivative of E with regard to j-th node's net input u_j, we obtain equation 3-6 where e_j is the prediction error:

$$\frac{\varphi E}{\varphi u_j} = y_j - t_j := e_j \tag{3-6}$$

where t_j = 1 (j = j^*), i.e., t_j will only be 1 when the j-th node is the actual output word, otherwise t_j = 0. Note that this derivation is the prediction error e_j of the output layer. Next we take the derivative on $word_{ij}$ to obtain the gradient on the hidden $\rightarrow$ output weights.

$$\frac{\varphi E}{\varphi word'_{ij}} = \frac{\varphi E}{\varphi u_j} . \frac{u_j}{\varphi word'_{ij}} = e_j . h_i \tag{3-7}$$

Therefore, using stochastic gradient descent, we obtain the weight updating equation for hidden $\rightarrow$ output weights:

$$\mathbf{v'}_{word_j}^{(new)} = \mathbf{v'}_{word_j}^{(old)} - \eta \,.\, e_j \,.\, \mathbf{h} \mid \; j = 1, 2, ..., N. \tag{3-8}$$

where $\eta > 0$ is the learning rate, e_j = y_i - t_j, and h_i is the i-th node in the hidden layer; $\mathbf{word'}_j$ is the output vector of $word_j$.

Having obtained the update equations for **W'**, we can now move on to **W**. We take the derivative of E on the output of the hidden layer, obtaining

$$\frac{\varphi E}{\varphi h_i} = \sum_{j=1}^{V} \frac{\varphi E}{\varphi u_j} \,.\, \frac{u_j}{\varphi h_{ij}} = \sum_{j=1}^{V} e_j \,.\, word'_{ij} := EH_i \tag{3-9}$$

where h_i is the output of the i-th node of the hidden layer; u_j is defined in (3-2), the net input of the j-th node in the output layer; and e_j = y_j - t_j is the prediction error of the j-th word in the output layer. EH, a N-dim vector, is the sum of the output vectors of all words in the vocabulary, weighted by their prediction error.

Next we should take derivative of E on **W**. The hidden layer performs a linear computation on the values from the input layer. Expanding the vector notation in (3-1) we get equation 3-10:

$$h_i = \sum_{k=1}^{V} x_k \,.\, word_{ki} \tag{3-10}$$

Now we can take the derivative of E with regard to **W**, obtaining equation 3-11:

$$\frac{\varphi E}{\varphi word_{ij}} = \frac{\varphi E}{\varphi h_i} . \frac{h_i}{\varphi word_{ki}} = EH_i . x_k \tag{3-11}$$

from which we obtain $V \times N$ matrix. Since only one component of **x** is non-zero, only row of $\frac{\varphi E}{\varphi \mathbf{W}}$ is non-zero, and the value of that row is EH, a N-dim vector. We obtain the update equation of **W** as:

$$\mathbf{v}_{word_j}^{(new)} = \mathbf{v}_{word_j}^{(old)} - \eta \,.\, EH \tag{3-12}$$

where $\mathbf{v}_{wordi}$ is a row of **W**, the "input vector" of the only context word, and is the only row of **W** whose derivative is non-zero.

After Word2Vec training, the softmax layer stores final probability values. These values can be accessed by passing the input and output words. The artificial neural

network training process shown in this section considers all vocabulary words in the calculation performed between the hidden and output layers. It is important to emphasize that this process is not feasible due to computational complexity.

Therefore, the process should be optimized because normalization is computationally expensive. Thus, to achieve a computational efficiency, there are two methods to handle this: Hierarchical Softmax and Negative Sampling.

Hierarchical Softmax uses a binary Huffman tree to represent all words in the vocabulary for fast training. The V words must be leaf nodes of the tree. For each leaf node, there exists a unique path from the root to the node; and this path is used to estimate the probability of the word represented by the leaf node. Negative Sampling is another way to handle Word2Vec training complexity. This method is more straightforward than Hierarchical Softmax, because we can just sample some of the nodes which we compute scores for [65].

CBOW architecture allows us to define the number of input words: one-word or multi-word. The amount of words is defined according to the size of the context or window of words neighboring the word target. The difference between one-word represented in Figure 3.2 and multi-word context represented in Figure 3.3 is instead of directly copying the input vector of the input context word, the CBOW model takes the average of the vectors of the input context words, and use the product of the input to hidden weight matrix and the average vector as the output.

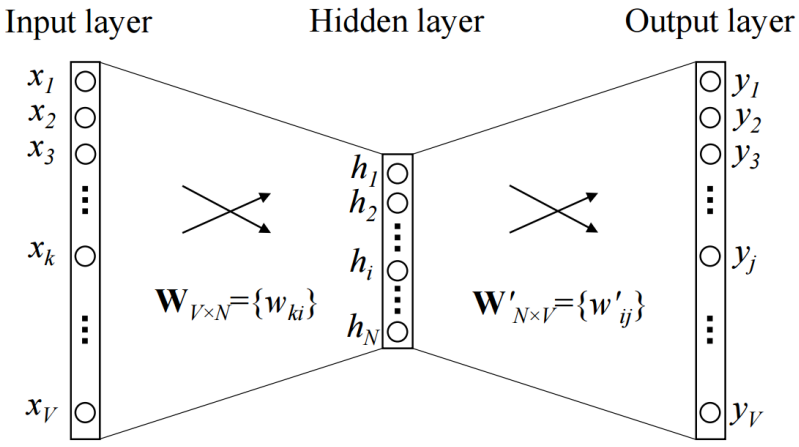


Figure 3.2: *CBOW model with a one-word context setting [65]*

The new value of h in CBOW model with a multi-word context setting is defined in 3-13 and 3-14. Figure 3.3 shows the CBOW model with a multi-word context setting.

$$\mathbf{h} = \frac{1}{C}\mathbf{W}.(X_1 + X_2 + ... + X_C) \tag{3-13}$$

$$\mathbf{h} = \frac{1}{C}.(\mathbf{v}_{w1} + \mathbf{v}_{w2} + ... + \mathbf{v}_{wC}) \tag{3-14}$$

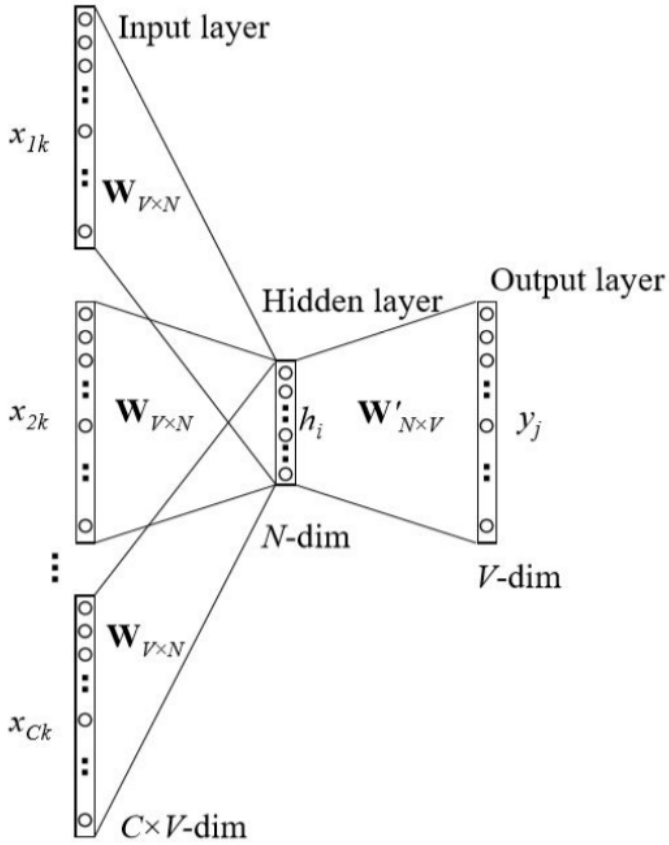


Figure 3.3: *CBOW model with a multi-word context setting [65]*

The skip-gram model is introduced in Mikolov[57]. Figure 3.4 shows the skip-gram model. It is the opposite of the CBOW model. The target word is now at the input layer, and the context words are on the output layer[65].

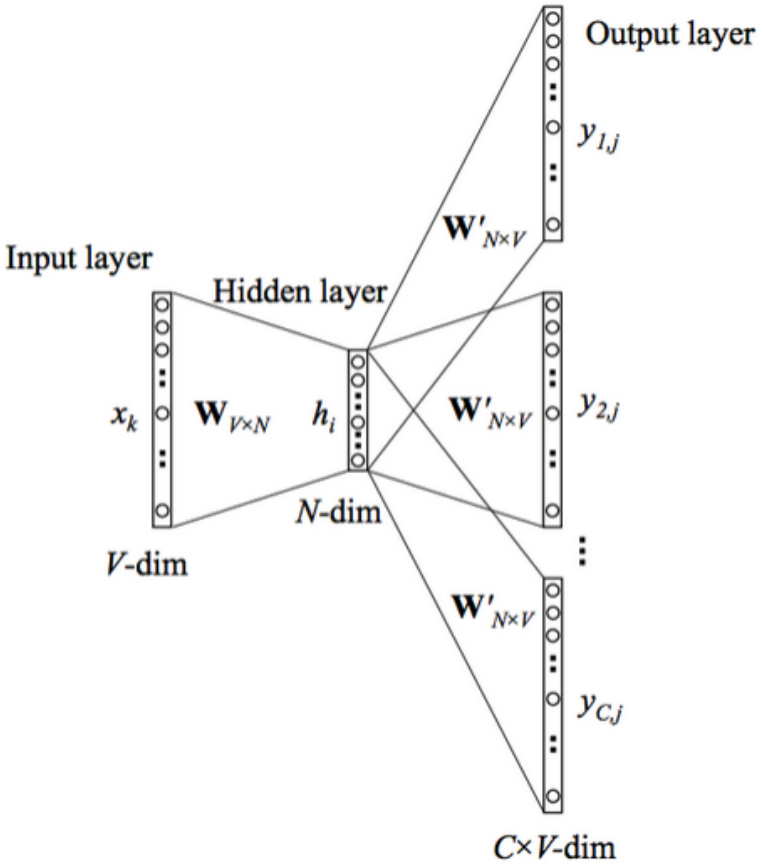


Figure 3.4: *Skip-gram model [65]*

In natural language processing demands, producing a good measure of the similarity of two documents depends Word2Vec architecture suitable choice. Besides, creating a good measure of the similarity of two documents can be a helpful building block. Kusner et al. proposes Word Mover's Distance (WMD) [28]. They adapted the earth move's distance to the space of documents: the distance between two documents is given by the total amount of bulk required to transfer the words from one side into the other, calculated by the distance the words need to move. So, starting from a measure of the distance between different words, we can get a document-level distance.

Word Mover's distance (WMD) uses the word2vec vector embeddings of words to assess the distance between two text documents. It calculates the cumulative sum of minimum distance each embedded word of one document needs to move to the closest embedded word of another document. Thus, even when the two documents have no words in common, the WMD is able to accurately measure the similarity between the two documents. Thus, documents sharing many words (the same or related word) end up having smaller distances compared to documents with very dissimilar words.

Formally the WMD is described as follows. We are given a word embedding matrix $\mathbf{X} \in \mathbb{R}^{d\times n}$ for a vocabulary of n words. Let $\mathrm{x}_i \in \mathbb{R}_d$ be the representation of the i^{th}

word and $\mathbf{d}^a$, $\mathbf{d}^b$ be the n-dimensional normalized bag-of-words (BOW) vectors for two documents. Also, let d_i^a be the number of times word i appears in $\mathbf{d}^a$, normalized over all words in $\mathbf{d}^a$. The WMD works with a 'transport' matrix $\mathbf{T} \in \mathbb{R}^{n \times n}$, in which $\mathbf{T}_{ij}$ describes how much of d_i^a should be transported to d_j^b [28].

3.4 Summary

In this chapter, we present different ways of representing documents and words as vectors. These vectors allow us to calculate the similarity between documents or words. In the next two chapters, we present our proposals to improve the performance of both the similarity search and embedding generation by exploiting parallel processing.

CHAPTER 4

Parallel approaches to Document Similarity Search and TOP-K Applications

The similarity search for textual data is commonly performed through a k nearest neighbor search in which pairs of document vectors are compared and the k most similar are returned. For this task, in this chapter, we propose FaSST-kNN, a fine-grain parallel algorithm, that applies filtering techniques based on the most common important terms of the query document using tf-idf.

4.1 k-NN

The k nearest neighbors (kNN) is an optimization problem for finding the closest k elements in metric spaces. Many techniques have used to tackle this problem. One approach to solving the kNN problem is an exhaustive search technique (also known as brute-force) to find the nearest neighbors of a point. In this technique, all distances between a query q and the points in metric space are computed. The next step is sorting the computed distances and selecting the k elements corresponding to the k smallest distances. The disadvantage of this approach is the high computational cost: $\mathcal{O}(nd)$ for calculating the n distances and $\mathcal{O}(nlogn)$ for sorting them.

Several kNN algorithms have been proposed to reduce computation cost [18, 19, 39, 47, 73, 59]. In general, the idea is to reduce the number of distances calculated. Some algorithms apply a method to partition the data points in a hierarchically way. In this category, we can use a tree-based data structure as known as kd-tree. This approach allows calculating distances within nearby space [18]. The increase in the cost of preprocessing is drawback's tree structured approach. Furthermore, for sufficiently high dimension, this method ends up having to search many additional nodes and offer little improvement over brute-force exhaustive searching.

Find only approximate nearest neighbors is an approach to achieving more efficiency. However, it sacrifices some accuracy or precision. Instead, to compute the exact nearest neighbors, an approximate method prefers to compute neighbors that are

close enough to the query item. Locality-sensitive hashing (LSH) is one of the popularly used approximate methods. LSH uses a family of hash functions to cluster nearby items into bins with a high probability [60]. Query objects are hashed into one bin whose items are used as potential candidates for the final results. The search time is sublinear in the collection data size. However, the bin sizes are usually broad enough to require an exact kNN search in the chosen bin. Besides, to scale, approximate methods sacrifice effectiveness using an approximated kNN solution.

4.2 TOP-k Applications

Searching is one of the critical fundamental problems in computer science, present in practically all applications. Most of the new searching algorithms were created with the traditional notion of exact search, i.e., it consisted of finding an element whose identifier matched exactly to a given search key. Nowadays, a high number of applications have to work with databases containing mostly unstructured data. In this scenario, it is not always possible to define important search keys for each database element. Besides, the rise of high dimensional big data from different data sources (e.g. images, sounds, text), and the lack of a natural ordering among dimensions made it hopeless to search using classical exact search techniques hierarchically. In modern information retrieval systems, the queries usually ask for relevant objects, which are related, to a given one, whereas comparison for exact identity is exceptional. Hence, the focus of searching has been changed to similarity search.

Besides that, with the expansion of e-commerce, recommender systems are attracting more and more attentions. E-commerce web sites are based on users' purchasing and browsing information, and then they recommend relevant products to the users. What will be recommended in the system is called item, which can be books, movies or restaurants, etc. Items of the same kind make an item set.

The principle of recommendation is estimating the candidate item set for a certain user, and then the item with the highest prediction value will be recommended to him or her. The most widely used recommending technology is Collaborative Filtering (CF). The principle most CF algorithms is creating a user-item rating matrix, then predicting how many users u may like an unknown item i, by analyzing a group of users who have the similar interest with u [33]. Nevertheless, CF has a significant drawback: the user-item rating matrix is very sparse, the accuracy of using CF will be significantly reduced. Therefore, Top-K recommender system has been used to tackling this disadvantage.

4.3 Related works

4.3.1 GT-kNN - the base algorithm of FaSST-kNN

The proposed parallel implementation, called GPU-based Textual kNN (GT-kNN), greatly improves the k nearest neighbors search in textual datasets[11]. The solution efficiently implements an inverted index in the GPU, by using a parallel counting operation followed by a parallel prefix-sum calculation, taking advantage of Zipf's law, which states that in a textual corpus, few terms are common, while many of them are rare [51]. This makes the inverted index a good choice for saving space and avoiding unnecessary calculations. At query time, this inverted index is used to quickly find the documents sharing terms with the query document. This is made by constructing a query index which is used for a load balancing strategy to evenly distribute the distance calculations among the GPU's threads. Finally, the k nearest neighbors are determined through the use of a truncated bitonic sort to avoid sorting all computed distances. Next we present a detailed description of these steps.

Creating the Inverted Index

The inverted index is created in the GPU memory, assuming the training dataset fits in memory and is static. Let $\mathscr{V}$ be the vocabulary of the training dataset, that is the set of distinct terms of the training set. The input data is the set $\mathscr{E}$ of distinct term-documents (t,d), pairs occurring in the original training dataset, with $t \in \mathscr{V}$ and $d \in \mathbb{D}_{train}$. Each pair $(t,d) \in \mathscr{E}$ is initially associated with a term frequency tf, which is the number of times the term t occurs in the document d. An array of size $|\mathscr{E}|$ is used to store the inverted index. Once the set $\mathscr{E}$ has been moved to the GPU memory, each pair in it is examined in parallel, so that each time a term is visited the number of documents where it appears (document frequency - df) is incremented and stored in the array df of size $|\mathscr{V}|$. A parallel prefix-sum is executed, using the CUDPP library [68], on the df array by mapping each element to the sum of all terms before it and storing the results in the *index* array. Thus, each element of the *index* array points to the position of the corresponding first element in the *invertedIndex*, where all (t,d) pairs will be stored ordered by term. Finally, the pairs (t,d) are processed in parallel and the *frequency-inverse document frequency* tf-$idf(t,d)$ for each pair is computed and included together with the documents identification in the *invertedIndex* array, using the pointers provided by the *index* array. Also during this parallel processing, the value of the norm for each training document, which is used in the calculus of the cosine or Euclidean distance, is computed and stored in the *norms* array. Algorithm 4.1 depicts the inverted index creation process.

Algorithm 4.1: *CreateInvetedIndex*(E)

input : term-document pairs in $E[0..|\mathscr{E}|-1]$.
output: df, *index*, *norms*, *invertedIndex*.

1 array of integers $df[0..|\mathscr{V}|-1]$ `// document-frequency array, initialized with zeros.`
2 array of integers $index[0..|\mathscr{V}|-1]$.
3 array of floats $norms[0..|\mathbb{D}_{train}-1|]$.
4 $invertedIndex[0..|\mathscr{E}|-1]$ `// the inverted index`
5 Count the occurrences of each term in parallel on the input and accumulates in df.
6 Perform an exclusive parallel prefix sum on df and stores the result in *index*.
7 Access in parallel the pairs in E, with each processor performing the following tasks:
8 **begin**
9 Compute the tf-idf value of each pair.
10 Accumulate the square of the tf-idf value of a pair (t,d) in $norms[d]$.
11 Store in *invertedIndex* the entries corresponding to pairs in E, according to *index*.
12 **end**
13 Compute in parallel the square root of the values in array *norms*.
14 **Return** the arrays: *count*, *index*, *norms* and *invertedIndex*.

Figure 4.1 illustrates each step of the inverted index creation for a five documents collection where only five terms are used. If we take t_2 as an example, the index array indicates that its inverted document list (d_2, d_4) starts at position 3 of the *invertedindex* array and finishes at position 4 (5 minus 1).

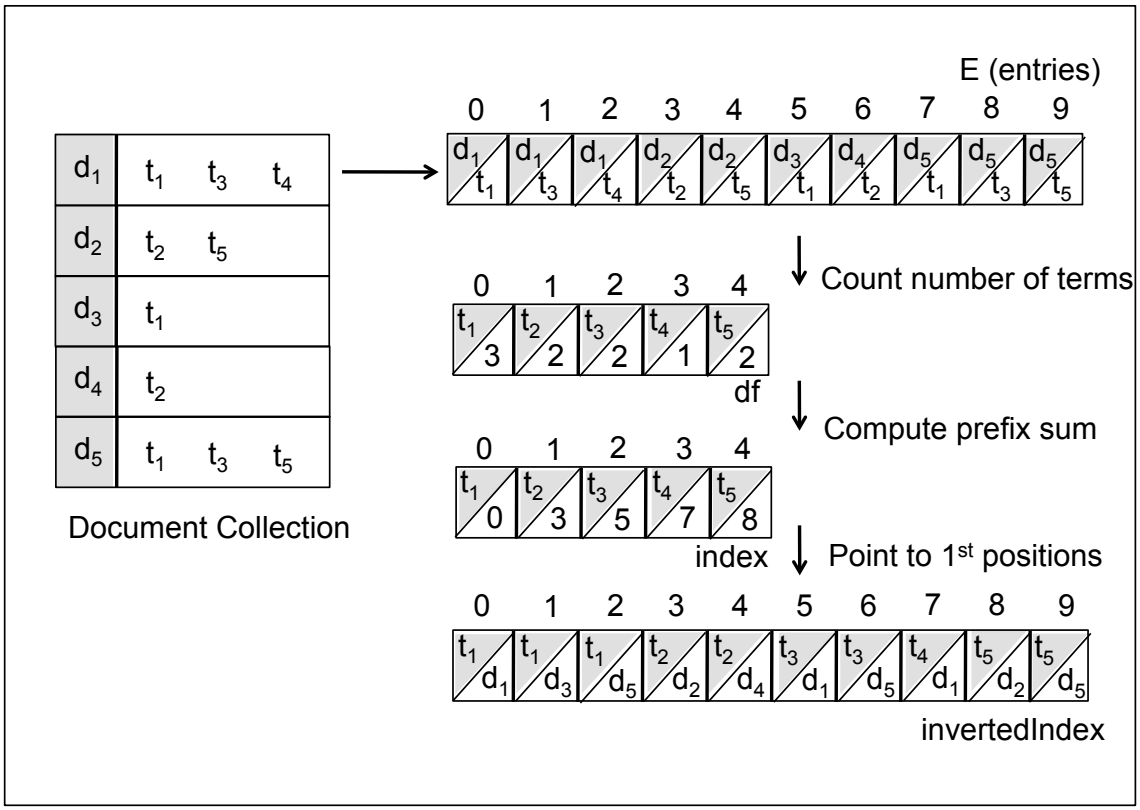


Figure 4.1: *Creating the inverted index*

Calculating the distances

Once the inverted index has been created, it is now possible to calculate the distances between a given query document q and the documents in $\mathbb{D}_{train}$. The distances computation can take advantage of the inverted index model, because only the distances between query q and those documents in $\mathbb{D}_{train}$ that have terms in common with q have to be computed. These documents correspond to the elements of the *invertedIndex* pointed to by the entries of the *index* array corresponding to the terms occurring in the query q.

The obvious solution to compute the distances is to distribute the terms of query q evenly among the processors and let each processor p access the inverted lists corresponding to terms allocated to it. However, the distribution of terms in documents of text collections is known to follow approximately the Zipf Law. This means that few terms occur in large amount of documents and most of terms occur in only few documents. Consequently, the sizes of the inverted list also vary according to te Zipf Law, thus distributing the work load according to the terms of q could cause a great imbalance of the work among the processors.

In this work besides using an inverted index to boost the computation of the distances, they also propose a load balance method to distribute the documents evenly among the processors so that each processor computes approximately the same number of distances [11]. In order to facilitate the explanation of this method, suppose that we concatenate all the inverted lists corresponding to terms in q in a logical vector $E_q = [0..|E_q|-1]$, where $|E_q|$ is the sum of the sizes of all inverted lists of terms in q. Considering the example in Fig. 4.1 and supposing that q is composed by the terms t_1, t_3 and t_4, the logical vector E_q would be formed by the following pairs of the inverted index: $E_q = [(t_1,d_1), (t_1,d_3), (t_1,d_5), (t_3,d_1), (t_3,d_5), (t_4,d_1)]$ and $|E_q|$ equals to six.

Given a set of processors $\mathscr{P} = \{p_0, \cdots p_{|\mathscr{P}|-1}\}$, the load balance method should allocate elements of E_q in intervals of approximately the same size, that is, each processor $p_i \in \mathscr{P}$ should process elements of E_q in the interval $[i\lceil\frac{|E_q|}{|\mathscr{P}|}\rceil, min((i+1)\lceil\frac{|E_q|}{|\mathscr{P}|}\rceil - 1, |E_q| - 1)]$. Consider the example stated above, and suppose that the set of processors is $\mathscr{P} = \{p_0, p_1, p_2\}$. Thus elements of E_q with indices in the interval $[0,1]$ would be assigned to p_0, indices in $[2,3]$ would be processed by p_1 and indices in $[4,5]$ would be processed by p_2.

Since each processor knows the interval of the indices of the logical vector E_q it has to process, all that is necessary to execute the load balancing is a mapping of the logical indices of E_q to the appropriate indices in the inverted index (array *invertedIndex*). In the case of the example associated to Fig. 4.1, the following mappings between logical indices and indices of the invertedIndex array must be performed: $0 \rightarrow 0$, $1 \rightarrow 1$, $2 \rightarrow 2$, $3 \rightarrow 5$, $4 \rightarrow 6$ and $5 \rightarrow 7$. Each processor executes the mapping for the indices in the interval corresponding to it and finds the corresponding elements in the invertedIndex

array for which it has to compute the distances to the query.

Let $\mathscr{V}_q \subset \mathscr{V}$ be the vocabulary of the query document d. The mapping proposed in this work uses three auxiliary arrays: $df_q[0..|\mathscr{V}_q|-1]$, $start_q[0..|\mathscr{V}_q|-1]]$ and $index_q[0..|\mathscr{V}_q|-1]$. The arrays df_q and $start_q$ are obtained together by copying in parallel $df[t_i]$ to $df_q[t_i]$ and $index[t_i]$ to $start_q[t_i]$, respectively, for each term t_i in the query q. Once the df_q is obtained, an inclusive parallel prefix sum on df_q is performed and the results are stored in $index_q$.

Algorithm 4.2: *DistanceCalculation*$(invertedIndex, q)$

input : *invertedIndex*, df, *index*, *query* $q[0..|\mathscr{V}_q|-1]$.
output: distance array $dist[0..|\mathbb{D}_{train}|-1]$ initialized according to the distance function used.

1 array of integers $df_q[0..|\mathscr{V}_q|-1]$ initialized with zeros
2 array of integers $index_q[0..|\mathscr{V}_q|-1]$
3 array of integers $start_q[0..|\mathscr{V}_q|-1]$
4 **for** *each term* $t_i \in q$, *in parallel* **do**
5 $\quad df_q[i] = df[t_i]$;
6 $\quad start_q[i] = index[t_i]$;
7 **end**
8 Perform an inclusive parallel prefix sum on df_q and stores the results in $index_q$
9 **foreach** *processor* $p_i \in \mathscr{P}$ **do**
10 $\quad$ **for** $x \in [i\lceil\frac{|E_q|}{|\mathscr{P}|}\rceil, min((i+1)\lceil\frac{|E_q|}{|\mathscr{P}|}\rceil - 1, |E_q|-1)]$ **do**
$\quad\quad$ `// Map position` x `to the correct position` *indInvPos* `of the` *invertedIndex*
11 $\quad\quad pos = min(i : index_q[i] > x)$;
12 $\quad\quad$ **if** $pos = 0$ **then**
13 $\quad\quad\quad p = 0; offset = x$;
14 $\quad\quad$ **else**
15 $\quad\quad\quad p = index_q[pos-1]; offset = x - p$;
16 $\quad\quad$ **end**
17 $\quad\quad indInvPos = start_q[pos] + offset$
18 $\quad\quad$ uses $q[pos]$ and $invertedIndex[indInvPos]$ in the partial computation of the distance between q and the document associated to $invertedIndex[indInvPos]$
19 $\quad$ **end**
20 **end**

Algorithm 4.2 shows the pseudo-code for the parallel computation of the distances between documents in the training set and the query document. In lines 4-7 the arrays df_q and $start_q$ are obtained. In line 8 the array $index_q$ is obtained by applying a parallel prefix sum on array df_q. Next, each processor executes a mapping of each position x in the interval of indices of E_q associated to it to the appropriate position of the invertedIndex. This mapping is described in lines 10-17 of the algorithm. Then, the mapped entries of the inverted index are used to compute the distances between each document associated with these entries and the query.

Figure 4.2 illustrates each step of Algorithm 4.2 for a query containing three terms, t_1, t_3 and t_4, using the same collection presented in the example of Figure 4.1. Initially, the arrays df_q and $start_q$ are obtained by copying in parallel entries respectively from arrays df and *index*, corresponding to the three query terms. Next a parallel prefix sum is applied to array df_q and the $index_q$ array is obtained. Finally the Figure shows the mapping of each position of the logical array E_q into the corresponding positions of the invertedIndex array.

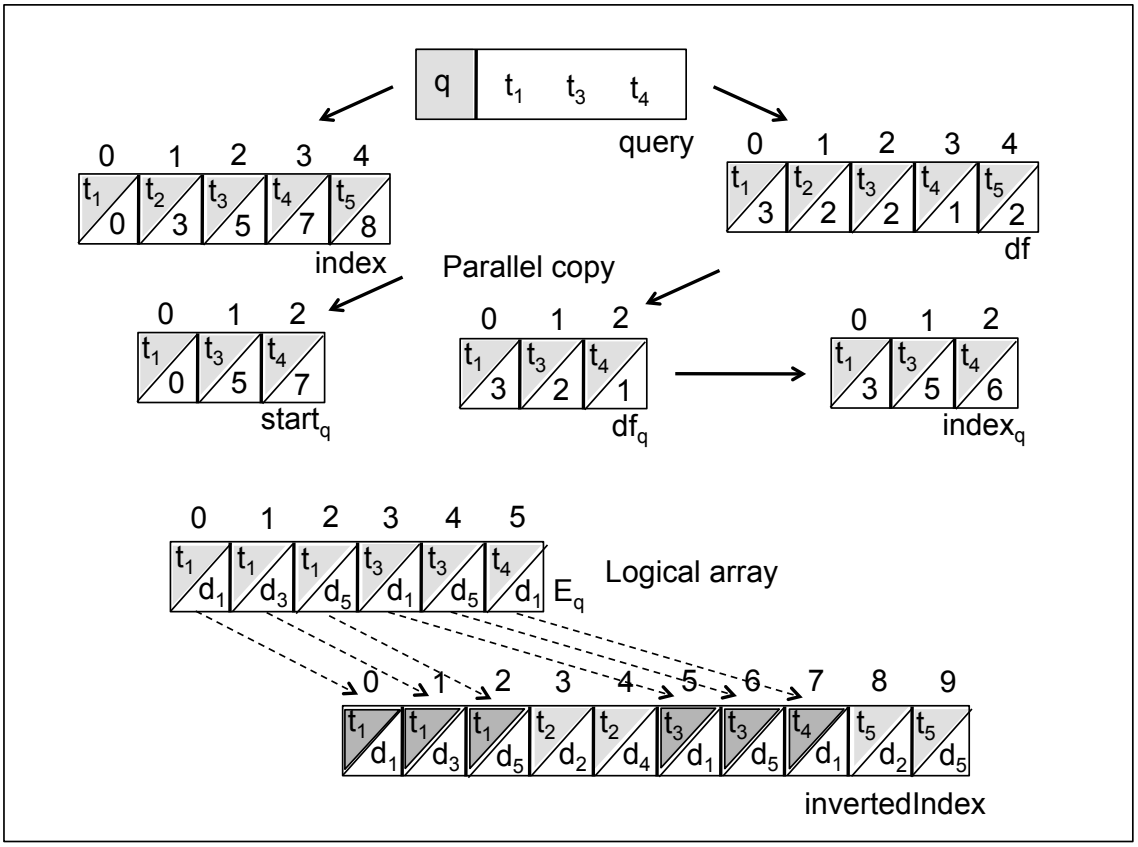


Figure 4.2: *Example of the execution of Algorithm 4.2 for a query with three terms.*

Finding the k Nearest Neighbors

With the distances computed, it is necessary to obtain the k closest documents. This can be accomplished by making use of a partial sorting algorithm on the array containing the distances, which is of size $|\mathbb{D}_{train}|$. For this, we implemented a parallel version of the Truncated Bitonic Sort (TBiS), which was shown to be superior to other partial sorting algorithms in this context [73]. One advantage of the parallel TBiS is data independence. At each step, the algorithm distributes elements equally among the GPU's threads avoiding synchronizations as well as memory access conflicts. Although the partial bitonic sort is $O(|\mathbb{D}_{train}| \log^2 k)$, worse than the best known algorithm which is $O(|\mathbb{D}_{train}| \log k)$, for a small k the ratio of $\log k$ becomes almost negligible. Their parallel TBiS implementation also uses a reduction strategy, allowing each GPU block to act independently from each other on a partition of array containing the computed distances. Results are then merged in the CPU using a priority queue.

Although the kNN algorithm can be applied broadly, it has some shortcomings. For large datasets and high dimensional space, its complexity $\mathcal{O}(nd)$ can easily become prohibitive. Moreover, if m succesive queries are to be performed, the complexity further

increases to $\mathcal{O}(mnd)$. This has motivated a number of parallel implementations of the kNN method over the last years. More recently, with the powerful and affordable GPU (Graphics Processing Units) many-core accelerators, some proposals have been presented to accelerate the kNN algorithm via a highly mutithreaded fine-grained data parallel approach.

Canuto's et al. work presents a new GPU-based implementation of kNN, called GPU-based Textual kNN (GT-kNN), specially designed for high-dimensional and sparse datasets. This algorithm allows a very fast and much more scalable meta-feature generation which allows one to apply this technique in large collections much faster. This solution does not use any filtering technique [11].

4.3.2 Other related algorithms

Johnson et al. proposed FAISS, an approximate kNN search algorithm for dense vectors [35]. It uses domain compression techniques, and inverted lists to build lookup tables for fast searching. It also provides an exact kNN search for dense datasets. It implements a brute force approach through matrix operations by using cuBLAS, a CUDA linear algebra library, and a novel GPU k-selection method that is applied during matrix multiplication. Although it is not suited for very high dimensions when the data is too large to fit in GPU memory the problem is tiled over query batches, while using pinned memory to overlap the data transfer and computation.

Gutiérrez et al. proposed GPU-SME-kNN (Scalable and Memory Efficient kNN) to accelerate kNN for large datasets with a memory efficient tiling scheme. Their method is able to incrementally select k neighbors by tiling the distance matrix and merging the k distances from the previous tile with a novel quicksort-inspired k-selection [26]. This work is not suited for sparse datasets.

Chen et al. proposed a GPU-based kNN that uses triangular inequality. It first creates clusters for the query and target sets, then applies an upper bound filter to discard distant target clusters. A second filtering is done on the remaining clusters by sorting the clusters and candidate points, in order to increase the effectiveness of filtering by triangle inequality. It also can adapt its algorithm according to input data [14].

Wang et al. proposed a similarity search for high-dimensional and sparse datasets, which implements novel LSH techniques that make it not require similarity computations and high memory. This work does not provide an exact solution, but it is the state-of-the-art GPU approximate kNN [80].

An approach that exploits fine granularity processing in an exact way is the G-kNN [64]. The G-kNN is a parallel algorithm of the kNN and was also implemented on GPUs. This algorithm stands out because it exploits high-dimensional and sparse

datasets, which is an advantage over other algorithms in the literature that parallelize *k*NN using GPU. This algorithm was developed to run in an environment with a single GPU. An indexing strategy based on a graph data structure was adopted and the algorithm was divided into two phases: the generation of the model and the classification of the documents. The first phase consists of two steps: calculating the distance between each test object and all training objects and ordering these distances. In the second phase, each test document is classified according to the nearest k training objects [64].

Matsumoto and Yiu proposed an exact solution by applying a fast sampling-based pruning method to compress distance matrices. In this work, samples are chosen at random from the full dataset. On a small sample, initial distance computation with a set of queries can be done quickly, with the *k*-th nearest neighbor extracted as the threshold value. Objects that have a distance from the query greater than the threshold are discarded. To balance parallelism and compression, queries are arranged into smaller batches. Besides that, they use a standard matrix-based approach to compute Euclidean (L2) distance between the data points and the query points [52].

Alewiwis et al. approach proposes the application of a new filtering technique that decreases the number of comparisons between the query set and the search set to find highly similar documents. In general, this method proposes a filtering technique known as prefix filtering, in which only the most important features are used to find highly similar documents. This work [2] does not provide an exact solution and its implementation is a coarse granularity solution. Another effort, not directly related to this work, use GPUs to accelerate the *k*NN search for low dimensional data with a *kd*-tree data structure [20].

4.4 Proposed work

Our proposal [1] differs from the above mentioned work in many aspects. First, it is an exact *k*NN solution but it does not use the brute-force approach. Since we deal with textual datasets, we avoid comparing the query with all documents in the collection, by creating an inverted index and quickly finding the documents sharing terms with the query document. This also save us a lot of space since the inverted index corresponds to a sparse representation of the data. Our proposal also handles well batches of queries, by simply processing one after another on a multi-GPU environment. In addition, our algorithm uses filtering techniques to discard documents that do not have important terms in common in relation to the query document. This strategy avoids unnecessary computing by further enhancing performance. Our goal is to achieve a high similarity value between the samples and the current query. At query time, we implement a threshold-based filtering by selecting samples (documents) that share terms with high TF-IDF values. The threshold is chosen among these samples, then all distances smaller than it can be pruned, while

the higher distances are compacted into a smaller array[1]. Our method uses the cosine as distance calculation. Finally, the k nearest neighbors are determined through the use of a radix sort algorithm on this smaller array.

4.5 A Parallel *k*NN Proposal

A fine-grained parallel algorithm takes advantage of data parallelism by processing individual items, terms of an individual document, in parallel. This greatly improves the k nearest neighbors search in textual datasets and can be easily mapped to modern highly threaded accelerators like manycore GPUs. Our implementation, called Fast Similarity Search for Text (FaSST-*k*NN), efficiently implements an inverted index taking advantage of Zipf's law, which states that in a textual corpus, few terms are common, while many of them are rare [51]. This makes the inverted index a good choice for saving space and avoiding unnecessary calculations. At query time, this inverted index is used to quickly find the documents sharing terms with the query document. This is made by constructing a query index which is used for a load balancing strategy to evenly distribute the distance calculations among the GPU's threads. The inverted index and distance calculation were based on GT-*k*NN [11]. Also at query time, it implements a new threshold-based filtering by selecting samples (documents) that share terms with high TF-IDF values. The threshold is chosen among these samples, then all distances smaller than it can be pruned. This is done on the GPU with a parallel compaction algorithm, which copies all distances greater than the threshold into a smaller array. Finally, the k nearest neighbors are determined through the use of a radix sort algorithm on this smaller array. In addition to exploiting intra-query parallelism, the solution also deals with inter-query parallelism, which allows the use of modern multi-GPU systems. Next, we present a detailed description of these steps.

4.5.1 Data Indexing

The data indexing process consists of creating an inverted index in the GPU memory, assuming the input dataset fits in memory and is static. Let $\mathscr{V}$ be the vocabulary of the input dataset, that is the set of distinct terms of the input set of documents $\mathbb{D}_{in}$. The input data is the set $\mathscr{E}$ of distinct term-documents (t,d), pairs occurring in the original dataset, with $t \in \mathscr{V}$ and $d \in \mathbb{D}_{in}$. Each pair $(t,d) \in \mathscr{E}$ is initially associated with a term frequency tf, which is the number of times the term t occurs in the document d. An array of size $|\mathscr{E}|$ is used to store the inverted index. Once the set $\mathscr{E}$ has been moved to the GPU

[1] Since the cosine distance is a similarity metric, the nearest distances are the higher ones.

memory, each pair in it is examined in parallel, so that each time a term is visited the number of documents where it appears (document frequency - df) is incremented and stored in the array df of size $|\mathcal{V}|$. A parallel prefix-sum is executed on the df array by mapping each element to the sum of all terms before it and storing the results in the *index* array. Thus, each element of the *index* array points to the position of the corresponding first element in the *invertedIndex*, where all (t,d) pairs will be stored ordered by the term. Finally, the pairs (t,d) are processed in parallel and the *term frequency-inverse document frequency* $tf-idf(t,d)$ for each pair is computed and included together with the documents identification in the *invertedIndex* array, using the pointers provided by the *index* array. Also during this parallel processing, the value of the norm for each input document is computed and stored in the *norms* array. These values are used later when calculating the cosine distances. [11].

4.5.2 *k* Nearest Neighbors Search

Given a query, the *k*NN search consists of two steps. First, the distances of the query (document) q to all input data (documents in $\mathbb{D}_{in}$) have to be computed. Then, the top k documents, that is, those closer to the query, are selected. The distances computation can take advantage of the inverted index model, because only the distances between query q and those documents in $\mathbb{D}_{in}$ that have terms in common with q have to be computed. These documents correspond to the elements of the *invertedIndex* pointed to by the entries of the *index* array corresponding to the terms occurring in the query q.

The obvious solution to compute the distances is to distribute the terms of query q evenly among the processors and let each processor p access the inverted lists corresponding to terms allocated to it. However, the distribution of terms in documents of text collections is known to follow approximately the Zipf's Law. This means that few terms occur in large amount of documents and most of terms occur in only few documents. Consequently, the sizes of the inverted list also vary according to the Zipf's Law, thus distributing the work load according to the terms of q could cause a great imbalance of the work among the processors.

The load balancing technique described in [11] is used to boost the computation of the distances, by distributing the documents evenly among the processors so that each processor computes approximately the same number of distances. After the distances are computed, the k closest documents to the query are selected. This is accomplished by making use of a sorting algorithm on the array containing the distances, which is of size $|\mathbb{D}_{in}|$. For this, we used the CUDA Thrust radix sort. Next, we describe our filtering proposal that reduce both this array's size and its sorting time.

4.5.3 Threshold-based Filtering

In order to increase the efficiency of the *k*NN computation, we propose a filtering techniques to decrease the sorting time of *k*NN candidates. By choosing a number of samples (documents) that have high similarity to the query, a threshold value can be chosen to discard all candidates whose similarity value are lower than it. In order to keep the *k*NN search exact, at least k samples need to be chosen, and the k-th smallest of those needs to be the threshold. This way, at most $|\mathbb{D}_{in}|$ - k candidates are filtered, in the best case where the true k are sampled. In the worst case, when the smallest of all distances is in the sample, no candidate would be discarded since there is no distance smaller than it.

For a query document x, we exploit the numerator from the cosine similarity function: $cos(x,y) = \frac{x \,.\, y}{\|x\|\|y\|}$, aiming to maximize it with documents y that share terms with high TF-IDF values. The main idea of our proposal is to sort the query's terms by TF-IDF in a descending order[2] and to execute one of the two possible proposed sampling methods. Also, to further increase the chance of selecting samples that will yield a higher similarity, we sort each inverted list by TF-IDF as well, as shown later in our proposal's flowchart.

Sampling Method #1

The first proposed method is based on choosing the documents' Ids from the inverted index list of the first query term, until k samples is completed. If it is not enough to complete the k required documents, the documents of the list of the next term are chosen; if it still does not complete, the choice is performed randomly. A set data structure is used to ensure that the samples have distinct Ids. This also applies for sampling method #2.

Sampling Method #2

The second proposed method receives a parameter T, which is the number of terms to be used. Then, the documents' Ids are chosen by doing a round-robin on the T inverted lists of the first T terms of the query, until it completes the required number of samples; if it is not enough, the documents in the list of the next term $T+i$ are chosen, like in the first method; if it still does not complete the sample size, it chooses randomly. For both methods, the random sampling only happens when the query's terms corresponding inverted lists has few documents, or the query itself has few terms.

[2]Meaning the first terms are likely to contribute more to the final cosine similarity value.

4.5.4 Fast Similarity Search for Text (FaSST-*k*NN)

Figure 4.3 shows the flowchart of our proposal FaSST-*k*NN, divided in four steps. First, the input data is read, then copied to GPU memory and is built as an inverted index. Then the inverted index lists are sorted on the GPU by using two stable sorts by key, an efficient approach to sort many sub arrays (inverted lists). One with the term id as the key, then one with the index TF-IDF as key. This sorted inverted index by TF-IDF is copied back to the CPU, where the samplings for queries are done. The next three steps are done for each query. The query is received, then sorted. Its terms are used in one of the proposed sampling methods, which will return a list of the chosen documents Ids, and copy it to the GPU. In the third step, the query is sent to the GPU, where the cosine distance is calculated against the inverted index, and written in a distance vector of size $|\mathbb{D}_{in}|$. By using the sample lists, only the chosen distances are copied to the CPU, where the threshold is returned with the QuickSelect *k*-selection algorithm. Finally, at step four, the threshold is sent to the GPU, where a GPU parallel compaction algorithm prunes all distances smaller than the threshold, and writes the distances greater than it at a smaller vector. This vector is sorted with Thrust radix sort, then the first *k* are copied to the CPU, and the corresponding document Ids are printed.

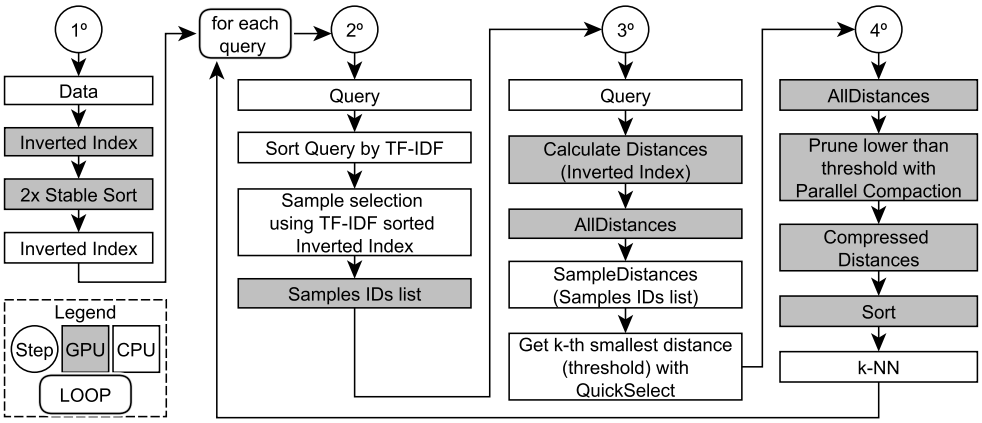


Figure 4.3: *FaSST-kNN flowchart.*

To facilitate the understanding of the algorithm, we show in the following figures what the algorithm does step-by-step. In Figure 4.4, we have 16 documents with Doc ID between 0 and 15. Each document with its terms and tf-idfs.

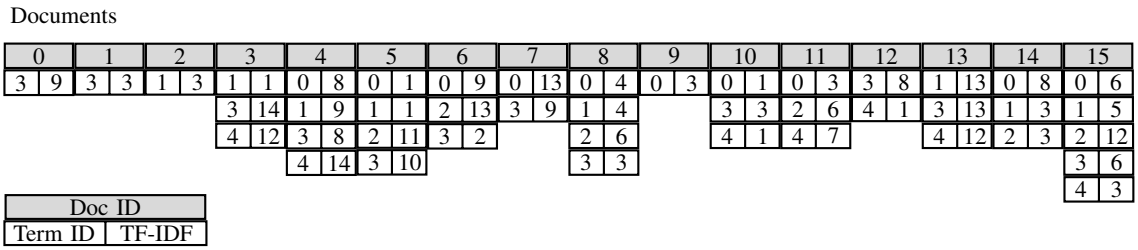


Figure 4.4: *Documents.*

In Figure 4.5, we have represented a query with terms 1, 3, and 2 with unordered tf-idf values. Each ID term has an inverted list containing which documents have it. At this step, the documents are not sorted by tf-idf.

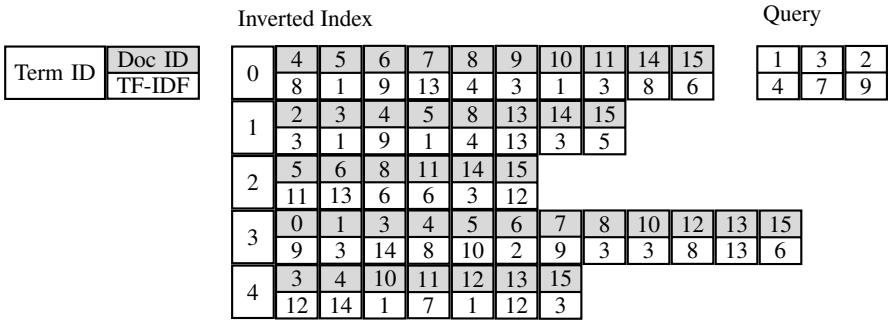


Figure 4.5: *Inverted Index and Query.*

In Figure 4.6, we have the inverted lists sorted in descending order. This operation aims to sample documents that have high common terms.

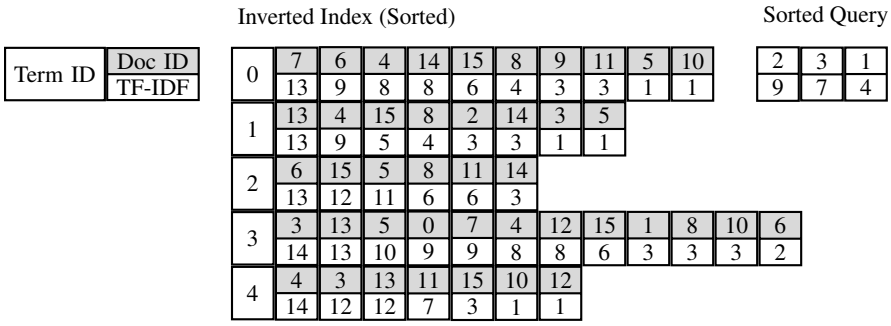


Figure 4.6: *Inverted Index and Query sorted by TF-IDF in descending order.*

In Figure 4.7, we assume that we want to get the 3 most similar documents. Our sample is defined as $2 \times k$, so let's initially select the documents that have the common term 2. At first, let's try to fill the list with all documents with term 2, but we might not have six documents with this term. In this case, we do have six documents with term

2. Otherwise, we would select the second most important query term, term 3 with tf-idf equal to 7. If we could not fill the rest of the list with documents containing this term, the list would be completed with random samples.

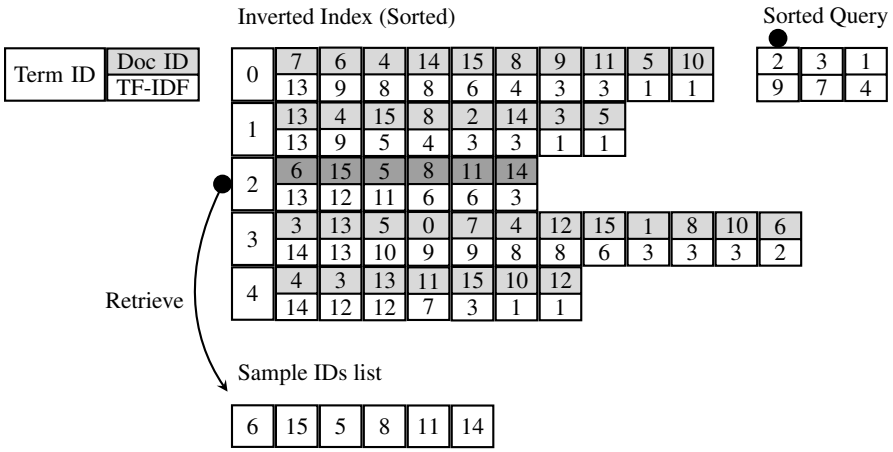


Figure 4.7: *Sampling method #1 with 6 samples.*

The next step, shown in Figure 4.8, is to calculate the scalar product between the query and the sample list documents considering term 2.

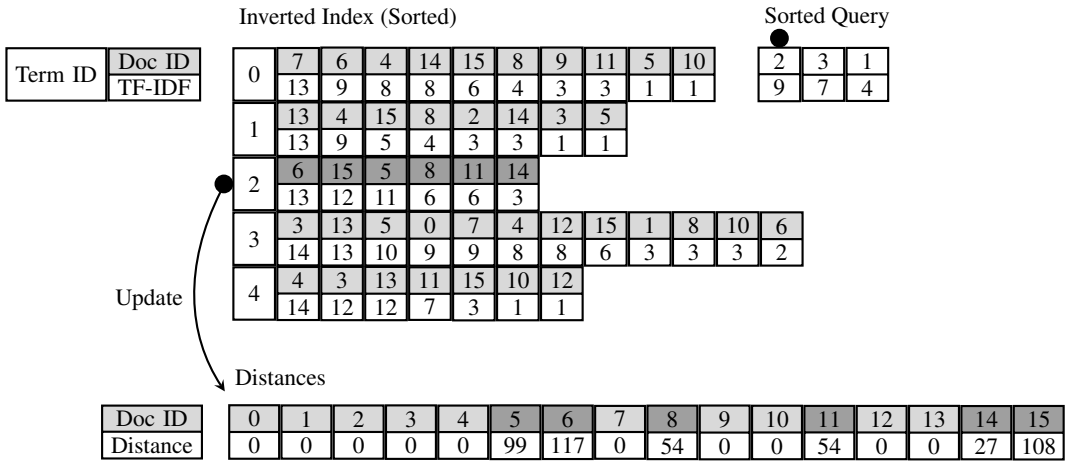


Figure 4.8: *Distance calculation. Dot product with term 2.*

We repeat the same process as in Figure 4.8 for terms 3 and 1 in the Figure 4.9 and Figure 4.10 respectively.

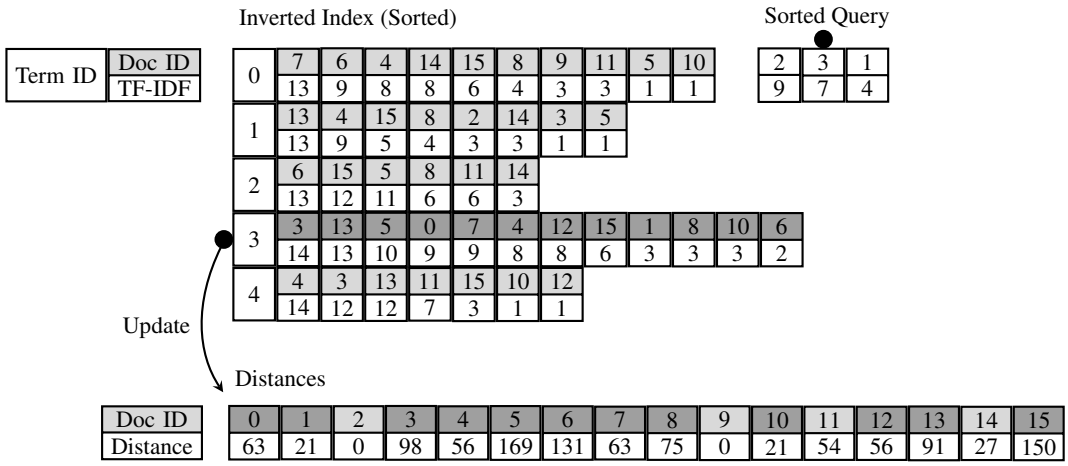


Figure 4.9: *Distance calculation. Dot product with term 3.*

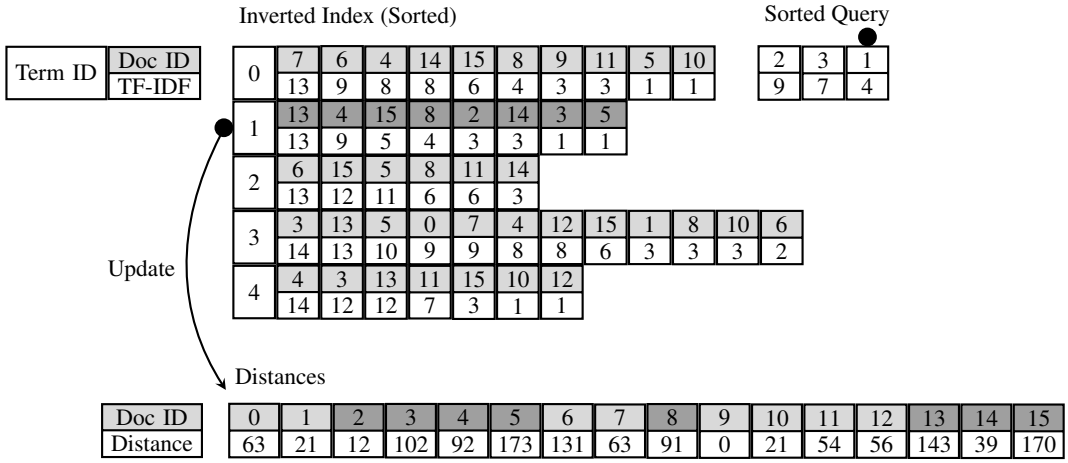


Figure 4.10: *Distance calculation. Dot product with term 1.*

After performing the same process for all query terms, the remainder of the distances is calculated as shown in Figure 4.11.

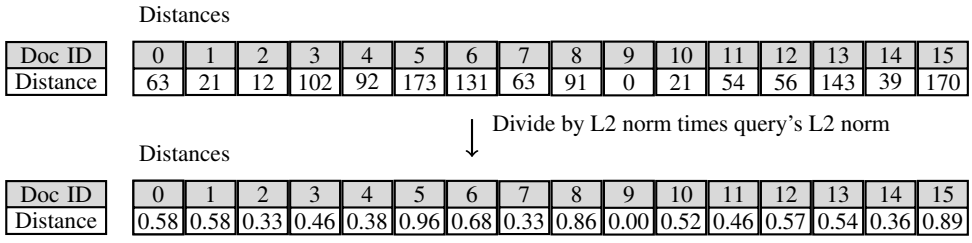


Figure 4.11: *Distance calculation. Dividing the dot product by the product of L2-norms.*

To accelerate matrix ordering of distances calculated in the previous step, we will generate a threshold to prune the matrix and generate a smaller matrix. Our method selects distance candidates considering only the documents with the most important terms

in common regarding the query as can be seen in Figure 4.7. The matrix with the candidate distances of size 2 *times k* is then sorted in ascending order. After that, a threshold will be selected as the *k*-th element, in the case of Figure 4.12, is the third element with value 0.68. From this threshold, documents having a distance greater than or equal to 0.68 will be selected and then sorted in descending order to select, in this case, the most similar *k* documents.

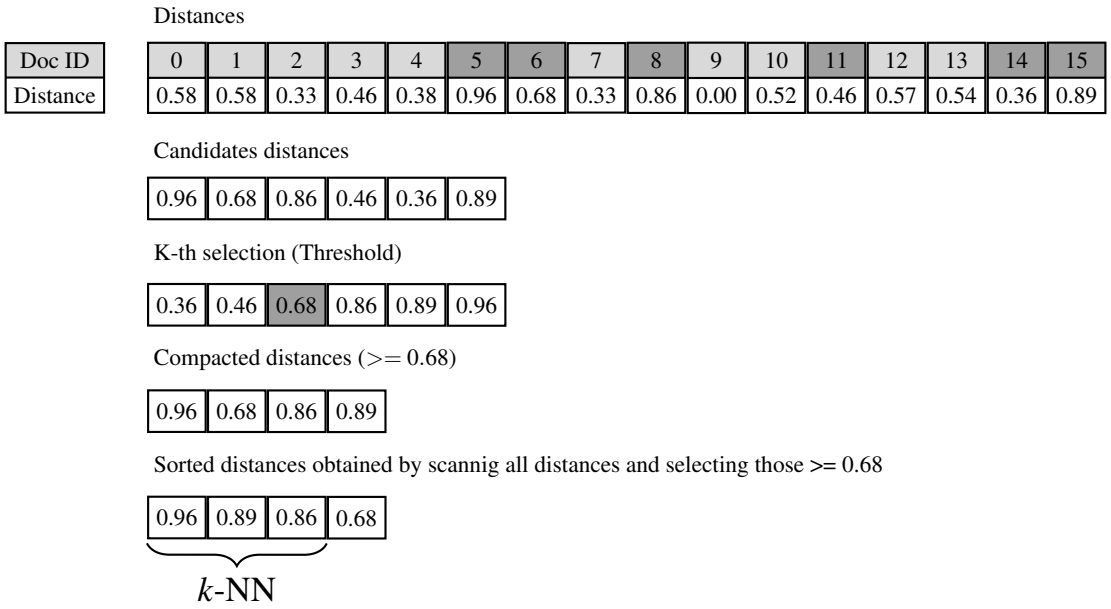


Figure 4.12: *Example of compaction phase with k=3 and samples=6*

Our method is accurate as it considers all calculated distances. The goal of the threshold is to eliminate as many elements as possible before sorting, which is a computationally expensive process. The calculated threshold may prune too little or not prune the matrix at all. To visualize this, imagine that the threshold is picked from the document with Doc ID 9 represented in the Figure 4.12 with distance zero. In this case, none element could be discarded, i.e., there would be no compaction.

In Figure 4.13, we have a situation where the Doc ID 3 document is not used in sampling. Note that in the final array of distances this document appears, but is not among the three most similar (k = 3).

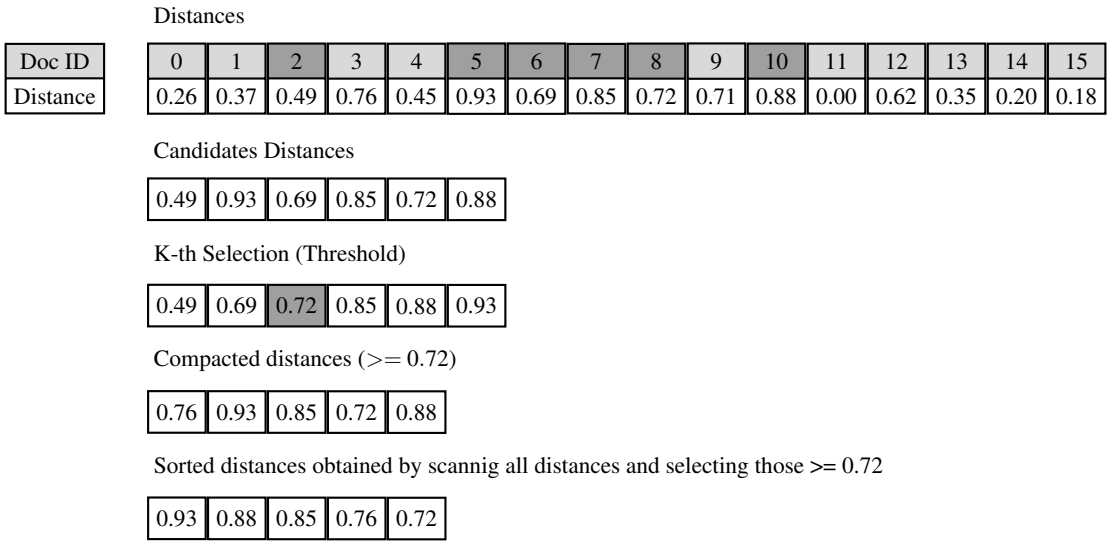


Figure 4.13: *Example 2 of compaction phase with k=3 and samples=6*

4.5.5 Multi-Query *k*NN Search

The FaSST-*k*NN algorithm[1] was designed having in mind a manycore architecture (accelerator) with (global) shared memory. It exploits data parallelism when processing a single query, and uses a single accelerator. It does that by making use of thousands of threads to index the dataset and to find the nearest neighbors of a single query document. However, in many situations, the *k*NN search has to be invoked repeatedly, to deal with the processing of many queries. The FaSST-*k*NN algorithm can handle that by processing the queries one after another, once the input data has been indexed. This streaming operation requires that the k nearest neighbors are returned before another query can be processed. In addition, a query specific memory allocation is needed for every query. Moreover, machines with more than one accelerator (GPU) can not take advantage of the extra computing power for the *k*NN search. This has motivated us to extend the FaSST-*k*NN to deal with multiple queries in a multi-GPU platform.

In the multi-GPU version, called SFaSST-*k*NN, task parallelism is exploited in addition to data parallelism. The data indexing step is performed by replicating the input data $\mathscr{E}$ in each of the g available GPUs and then, in parallel, creating g copies of the inverted-index. Thus, each GPU receives the same input and they all produce the same inverted-index in their memory. Next, each GPU performs the *k*NN search by dynamically requesting queries, so that there is no load imbalance between GPUs when queries sizes differ greatly. This is possible since the queries are completely independent of each other. Since the queries are of different size, we preallocate memory based on the biggest query, i.e. the document with the largest number of terms. This saves us a lot of time since GPU memory allocation can be very costly.

4.5.6 Scalable Fast Similarity Search for Text (SFaSST-*k*NN)

To increase the scalability of our FaSST-*k*NN algorithm, we implemented an inverted index distributed among GPUs. Step 1 as shown in Figure 4.3 has been adjusted to enable this implementation. While reading training data, a CPU thread takes care of distributing the data that is used to create the inverted index on existing GPUs. If we have *N* GPUs, then *N* input vectors (doc_id, term_id, and tf_idf tuples) will be allocated. Each document is read in a round-robin fashion as shown in Figure 4.14. Each document has its *ID* remapped between 0 and *N* / number of GPUs to have a contiguous vector. Maintaining the contiguous vector is important because it avoids vector gaps that would never be accessed, which would lead to more memory usage and worsen coalesced memory access. For each GPU is allocated a vector of equal size, that is, the final size of the CPU input vector. Each GPU creates its partial inverted index. The indexes are sorted and each CPU thread gets a copy of the partial indexes (only the index associated with the current GPU) to sample. Because sampling is done by document ID, using the partial index from another thread would give an error because it does not exist at the current index.

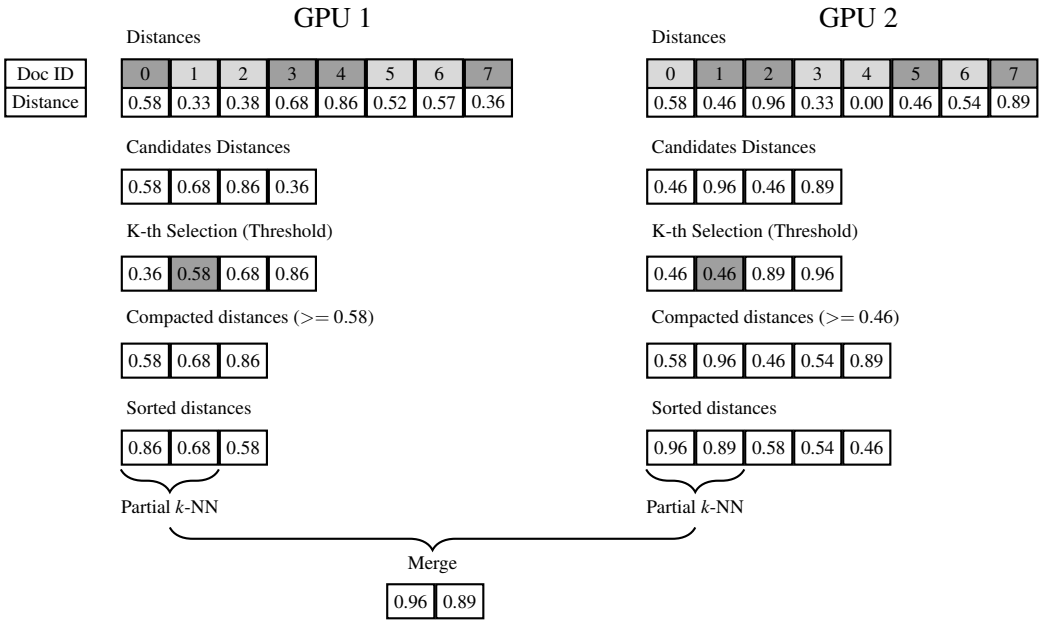


Figure 4.14: *Example of compaction phase with k=2, samples=4 and two distributed indexes.*

So during a query, a single CPU thread prepares it. The CPU threads sample in parallel on their partial inverted indexes and send the sample Ids to the GPU. After this process, the distance calculation is performed and then the filter is applied according to the threshold from the sample. As a result of applying the filter, we will have the partial *k*NN of each thread. After this, a merge is performed to generate the final *k*NN. Importantly, when the batch size is greater than 1, each CPU thread works in a different batch query, so when the batch size increases, the SFaSST gain is higher.

This new proposal differs from the FaSST-*k*NN in many aspects. First, it is an exact *k*NN solution but it does not use the brute-force approach. Since we deal with textual datasets, we avoid comparing the query with all documents in the collection, by creating an inverted index and quickly finding the documents sharing terms with the query document. This also save us a lot of space since the inverted index corresponds to a sparse representation of the data. Other advantage of our approach is that we do not have to rely on multiple queries to achieve high performance. A single query is enough to completely occupy the GPU and this is important since some applications may require a fast processing of a continuous streaming of data. However, our proposal also handles well batches of queries, by simply processing one after another on a multi-GPU environment. In addition, our algorithm uses filtering techniques to discard documents that do not have important terms in common in relation to the query document. This strategy avoids unnecessary computing by further enhancing performance. Our goal is to achieve a high similarity value between the samples and the current query. At query time, we implement a threshold-based filtering by selecting samples (documents) that share terms with high TF-IDF values. The threshold is chosen among these samples, then all distances smaller than it can be pruned, while the higher distances are compacted into a smaller array[3]. Our method uses the cosine as distance calculation. Finally, the k nearest neighbors are determined through the use of a radix sort algorithm on this smaller array.

4.6 Summary

Intensive use of *k*NN, combined with the high dimensionality and sparsity of textual data, makes it a challenging computational task. We have presented a fine-grained parallel algorithm and a very fast and scalable GPU-based approach for computing the top k nearest neighbors search in textual datasets. Different from other GPU-based *k*NN implementations, we avoid comparing the query document with all training documents. Instead, we use an inverted index in the GPU that is used to quickly find the documents sharing terms with the query document. Although the index does not allow a regular and predictable access to the data, a load balancing strategy is used to evenly distributed the computation among thousand threads in the GPU. Furthermore, we proposed a filtering technique that decreases the sorting time of *k*NN candidates. Our filtering method allows the removal of documents with similarity less than a threshold so that our algorithm spends less time in the sorting phase of the nearest k documents. Our proposal was extended to exploit multi-GPU platforms thus permitting the processing of multiple queries in parallel. We tested our approach in a very memory-demanding

[3] Since the cosine distance is a similarity metric, the nearest distances are the higher ones.

and time-consuming task which requires intensive and recurrent execution of *k*NN. The **FaSST-*k*NN** improved the top *k* nearest neighbors search by up to **60x** compared to a baseline. Improving the *k*NN performance will enhance several TOP-*K* applications. *k*NN complexity has motivated some parallel implementations of the *k*NN method over the last years, and we found new opportunities to improve them. Thus, we implement the FaSST-*k*NN and SFaSST-*k*NN, two fast GPU-based tool for computing the top *k* nearest in high dimensional and sparse data.

CHAPTER 5

Parallel approaches to accelerate word embedding generation with fine-grain parallelism

Word embedding is the collective name for a set of language modeling and feature learning techniques in natural language processing. Word2vec is a collection of related models that are used to produce word embeddings. Words or phrases from the vocabulary are mapped to vectors of real numbers in a low-dimensional space (typically of several hundred dimensions) relative to the vocabulary size. These word vectors are positioned in the vector space such that words that share common contexts in the corpus are located in proximity to one another in the space [57]. Word2Vec remains a popular choice for building word vectors due to their efficiency and simplicity. Although these fast algorithms are widely used, generating word embeddings are still too costly, which impacts negatively on time for conducting experiments in both Information Retrieval and Machine Learning applications. For this task, in this chapter, we present how to accelerate the generation of word embeddings used in the context of Automated Program Repair. In particular, the word embedding implemented is an extension of the well known Word2Vec with the CBOW model. Besides, we present how we can exploit fine-grained parallelism in this context and how we developed a new way of choosing negative samples that is beneficial for exploiting parallelism.

5.1 Related works

The first state-of-the-art algorithms for Word embeddings including Word2Vec have been parallelized for multi-core CPU architectures but are based on vector-vector operations that are memory-bandwidth intensive and do not efficiently use computational resources. The original Word2Vec was implemented this way.

To reduce generation time for word vectors, [Liu, 2014] implemented the optimization CBOW model in CUDA. He made an in-warp approach to CUDA architecture.

A WARP works with one word and updates the hidden layer of the artificial neural network in shared memory. Each WARP of thread updates the weights for one word of the sentence. It uses one warp to handle one data (one word), and 32 threads manage the parallelism in one data in a warp [48].

In [30], it was shown that the reuse of various data structures in the algorithm through the use of mini-batching, allows one to express the problem using matrix product operations producing good strong scalability. In [78], it was explored an unconventional training method to train networks without gradient descent steps.

In [25], they present BlazingText, a high performance distributed Word2Vec implementation that leverages massive parallelism provided by modern GPUs. They exploit GPU parallelism to discover the right balance between throughput and accuracy. That work replicated the matrices and processed disjoint parts of sentences in each GPU. Their work uses one thread block per sentence. Since the corpus is too large to fit in the GPU memory, this work uses disk stream to the GPU and several sentences are batch transferred to decrease the cost of data transfer between CPU and GPU. And at regular intervals, their algorithm produces a combination of matrix weights (one coming from each GPU) and transmits the new weights (new matrices) for each GPU to continue its iteration. According to the authors, there was no significant fall in accuracy with multi-GPU implementation. Their proposed implementation achieves near-linear scalability across multiple GPUs.

In [70], they propose a fast approximation method of a softmax function with a very large vocabulary using singular value decomposition (SVD). SVD-softmax targets fast and accurate probability estimation of the topmost probable words during inference of neural network language models. The proposed method transforms the weight matrix used in the calculation of the output vector by using SVD. The approximate probability of each word can be estimated with only a small part of the weight matrix using a few large singular values and the corresponding elements for most of the words.

In [72], the authors propose new methods to increase the speed of the Word2Vec on multi-core shared memory CPU systems, and on modern NVIDIA GPUs with CUDA. They perform this on multi-core CPUs by batching training operations to increase thread locality and to reduce accesses to shared memory. They propose new heterogeneous NVIDIA GPU CUDA implementations of both the skip gram hierarchical softmax and negative sampling techniques that utilize shared memory registers and in-warp shuffle operations for maximized performance.

Grave et al. proposes an approximate strategy to train neural network based language models over vast vocabularies efficiently. Their approach, called adaptive softmax, avoids the linear dependency on the vocabulary size by exploiting the unbalanced word distribution to form clusters that explicitly minimize the expectation of computational

complexity. Their approach further reduces the computational cost by employing the specificities of modern architectures and matrix-matrix vector operations[23].

The proposals of the related works can be summarized in reducing the weight matrices of the Word2Vec model; perform matrix multiplication operations rather than vector-matrix; perform operations on batch matrices in order to hide latency; to exploit low-level GPU resources (operations using in-warp functions) in order to increase scalability and reduce memory access. As shown in the next section, some of these ideas were used in our proposed approach. However, none of these works implement the word embedding proposed in [4]. That extension of Word2Vec is able to provide two metrics based on Word2Vec softmax output and embedded words distances. These embedding-based metrics are required to evaluate program patches that are candidates to fix a bug.

5.2 Parallel solution

Our algorithm is an evolution of one of Word2Vec's first manycore implementations focused on CBOW architecture. The implementation that served as the starting point for our application was not in scientific articles, but its source code served as a basis for other publications, and it is in a public repository [Liu, 2014]. This code is an adaptation of the original implementation of Mikolov, the creator of Word2Vec, to work with a single GPU. The methodology of this implementation consists of the following steps: (i) to fragment the input dataset (corpus) in sentences using one CPU process; (ii) to send sentences asynchronously to the GPU using pinned memory to speed up data transfer.

To reduce generation time for word vectors, [Liu, 2014] has implemented the optimization CBOW model in CUDA. He has made an in-warp approach to CUDA architecture [48]. A WARP [1] works with one word and updates the hidden layer of the artificial neural network in shared memory. Next we describe our proposal's workflow in Figure 5.2.

Our main contribution is to make the multi-GPU implementation, following the principles of the BlazingText article [25], but making necessary changes that guarantee the accuracy of the model and enable new approaches of parallelism in Word2Vec. A CPU thread is associated with each GPU, and our algorithm distributes sentences equally distributed for each one. Each GPU has its updated model and at the end of an iteration, they copy the arrays of the input and output models to the CPU. With multiple threads, the array is averaged and then sent back to all GPUs. BlazingText used the NCCL library for

[1] A WARP is the most basic unit of the scheduling of the NVIDIA GPU, i.e, it is the smallest executable unit of code or processes a single instruction over all of the threads in it at the same time or is the minimum size of the data processed in SIMD fashion. A WARP currently consists of 32 threads on NVIDIA hardware.

this, using direct transfer between GPUs. But this reduction and synchronization operation implemented in the CPU took less than 1% of the total time.

Initial experiments with the new multi-GPU implementation produced linear speedup, but the accuracy of the model was very low compared to Mikolov's original application. We have identified that the word per warp approach [Liu, 2014] has increased the conflict of updating the model matrices. We changed the WARP approach by a thread-block per word approach. As a result of this adjustment, the accuracy was very close to the ideal value generated by the original implementation of Word2Vec.

We implemented the reduction operation with partial reductions in each warp of the block with warp shuffle, and one more at the end applied on the partials in the shared memory. Reduction processing time decreased when we removed the random number variables from shared memory because there was a synchronization barrier for only one thread to update every time. Then it was implemented the possibility to execute multiple CPU threads for the same GPU, dividing the set of sentences again. It gave a small performance gain of approximately 5%. To improve accuracy, we used a vector for the persistence of the random numbers generated for each block, since for each sentence the value always starts from the same seed, which was the id of the block. With this vector, the value starts from where it had stopped.

To improve the accuracy of negative sampling, we have implemented a new strategy to select negative samples as can seen in Figure 5.1. Negative samples for a given word are the k less similar or farthest neighbor words (kFN - k - Farthest Neighbors). The metric used for this is the cosine similarity. We have applied the cosine similarity between the array vectors from all words to all[2]. This array stores the final vectors of words. First, in each GPU the L2 norm of each word/vector is calculated using a warp per word approach, and then a reduction with warp shuffle is applied.

[2]It is a traditional kNN but returns the farthest k vectors. It is not a version of kNN proposed in this research.

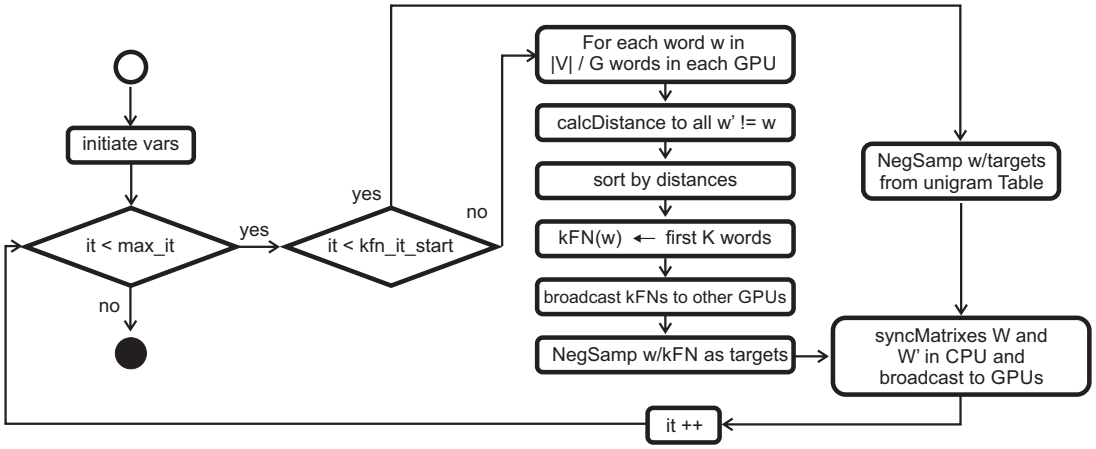


Figure 5.1: *The proposal workflow.*

Then each GPU calculates the *k*FN of a portion of all words, then joins the results in the CPU, and sends the complete set of *k*FNs to each GPU. We calculated the distances on the GPU, with a kernel launch per word, where each warp calculates the similarity of a word to the current target word. These distances are ordered with Thrust Radix Sort [8], and then the first (less similar) k words are copied to a vector where the *k*FNs of each word are stored. The *k*FN is applied after the synchronization of the matrices of the models. Then its result is only applied in the next iteration. During negative sampling method, samples are taken from the *k*FN vector of the current word.

As the arrays are randomly initialized, applying the *k*FN in the first iterations will generate neighbors that do not match what is expected. Therefore, a parameter was added to choose from which iteration the *k*FN would be used in place of the original sampling of negative sampling (based on the use of unigram table). Thus, the matrix would be closer to the ideal, allowing the *k*FN to find neighbors close to the real ones.

5.3 Application

Our purpose to accelerate the generation of word embeddings has enabled us to contribute a new way of assessing potential fixes in the area of Automated Program Repair (APR). As aforementioned, one of the applications that makes use of word embeddings is the APR. In such a field, a lot of code has to be analyzed in order to improve machine learning models for helping fix programs in an autonomous manner. Besides, a word embedding extracted from source code is relevant for APR when generating metrics useful to evaluate program patches. This task is performed during the process of generation and validation of program variants candidate to fix a bug.

In the last decades, software has gained increasingly more importance in several activities of our daily lives. Common tasks such as taking a taxi, as well as more complex ones, such as air traffic control, are widely supported by software, be it simple

or sophisticated. The maintenance of these software becomes indispensable to keep our society going on, while avoiding financial losses or fatal accidents, for instance. Meanwhile, according to [9], software maintenance is typically a costly activity, in which fixing buggy programs accounts for approximately 21% of all resources in the maintenance phase.

Besides, [81] argues that repairing a single failure may cost up to 200 working days and they amounted 1.7T USD in lossesfrom software failures in 2017 [79].

In this chaotic scenario, various approaches have been developed aiming at automatizing the software repair process and, consequently, reducing manual efforts of software maintenance. These works are leveraging an area called *Automated Program Repair* (APR) by applying several computational techniques with promising results. To name a few examples of APR techniques: [7], [43] applied Genetic Programming concepts; [37] and [54] used both symbolic execution and constraint solvers to produce software corrections; finally, [49] and [24] proposed different learning methods to fix incorrect fragments of code. In summary, they all exploit the space of solutions by automatically generating variants from an original buggy program and then evaluating them with some quality measure.

Despite recent findings, real-world APR applicability still continues to be a complex task, mainly because some of its essential steps are not trivial. For instance, evaluating variants (i.e. patches) generated by APR methods is a challenging task since different source codes (syntax) may share the same semantics. There is also the complexity of comparing partial results. The most common way to evaluate a patch is to rely on test cases or formal specifications. Typically, given an automatically generated program variant and a test suite, a possible evaluation is a weighted sum of how many test cases this variant passes [43]. Unfortunately, this kind of evaluation leads to plateaus, where patches with distinct source code have the same evaluation score as they pass in the same number of test cases, regardless of whether they are different test cases or not. Therefore, test case-driven methods are typically insufficient to establish a distinction over variants that pass the same number of test cases.

Assuming source codes are regular and predictable [16], we speculate that software naturalness can help the evaluation process of potential fixes for buggy programs. Some researchers have captured this naturalness of software through statistical models [63] and then used them for named entity recognition [71], coding standards checkers and suggesting accurate method and class names [3]. Due to good results of previous works using embedding words and sequence-to-sequence methods, we believe that these approaches, including Word2vec, can also be applied to modeling fix naturalness and then improve the evaluation process of the variants.

To the best of our knowledge, this is the first work which applies word em-

beddings in software source code to generate new metrics for evaluating potential fixes. To investigate this approach, we consider a corpus of fixes from six different programs (checksum, digits, grade, median, smallest, and syllables) from the IntroClass Benchmark [44]. Overall, our results show that Word2vec is a promising method to recognize the naturalness pattern of source code and, therefore, the derived metrics (Softmax output layer and distance between document vectors) can help the evaluation process of variants. Our experiments are done with C projects, but they can be made for any programming language. For this, we can train our system with a corpus formed by fixed codes (found in repositories of open source projects).

Our main contributions are:

- A method to statically evaluate variants generated by automated program repair methods;
- A methodology based on embedded words and sequence-to-sequence models to evaluate the naturalness of a potential fix;
- An experimental study using standard program repair benchmarks.

5.3.1 Approach

This section presents the proposed approach to evaluate variants generated by APR methods.

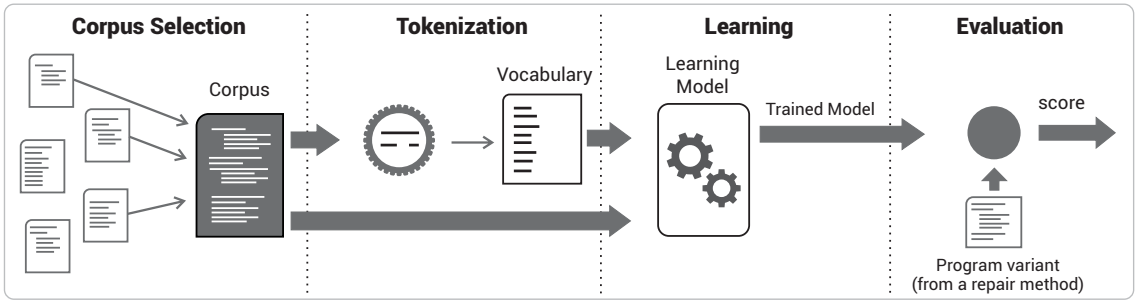


Figure 5.2: *Proposal's flowchart*

Our approach has three phases (Figure 5.2). The first phase, called Corpus Selection, consists in selecting the fix corpus, which is a reference for determining the semantic programming patterns of a developer community. The quality of the selected fixes has a critical impact on the performance of the proposal since the learning method uses this data to recognize the pattern.

The second phase, called Tokenization, defines how the source codes from the corpus are split and are used to create the vocabulary. The input to this phase is a corpus of fixes, syntax formatted and standardized. The output is a vocabulary with all recognizable tokens (classes).

The third phase, called Learning, builds a model able to predict the next token (target) given its context (set of tokens). For each pair (context token and target token) in a training fix code, Word2vec examines lots of pairs to effectively learn that "token X and Y often appear together" and "token X and W do not". The algorithm output is a matrix with word vectors (a trained model).

The fourth phase, called Evaluation, uses the embedded words or softmax layer output to calculate the score for each variant. After that, the highest score variant is chosen as the RefCode (Reference Code).

To perform the last phase, we adapted Word2Vec algorithm to obtain the probabilities in the output softmax layer after the artificial neural network training finish. Figure 3.2 shows the network with one-word or one-token context, the architecture used in this work. In our setting, the vocabulary size is V, and the hidden layer size is N. The nodes on adjacent layers are fully connected. The input vector is the one-hot encoded vector, which means that for a given output context word, only one node of $\{x_1, \ldots, x_V\}$ is 1, and all other nodes are 0 [65].

5.3.2 Related Work

Many works have focused on proposing and evolving techniques to address automated software repair. As we discussed before, almost all of these approaches are based on software specification, for example [21], [34], [5] or test cases [6], [17], [22], [53], [58], [61], [62], [76] and [77]. In other words, these techniques are vulnerable to partial specifications, pool test suites and time concerns, because typically it is necessary to run or, at least, simulate all tests for each variant under evaluation. Especially when evolutionary techniques are used, executing a large number of test cases may be infeasible or their absence may cause weakness in the fitness evaluation.

Our proposal performs a static analysis by applying the learned model, thus, it does not require to execute test case or even original program and variants for evaluating a patch.

Recently, some works proposed metrics or mechanisms to alleviate test cases dependence. In [31] is introduced a re-usability metric to help on repairing programs using fix ingredients from similar code extract from correct programs. To compound the metric, the proposal analyzes ASTs (Abstract Syntax Trees) and defines a level of similarity and difference between a buggy program and a fix candidate fragment of an external code. The authors argue that the similarity component is used to find fix ingredients whereas the difference component prevents picking up a code with the same bug under repairing. In opposition to our method, the metric re-usability is not able to rank a set of candidate patches, but only create a threshold through which it selects a set of promising variants.

The work proposed by [84] applies Genetic Programming to repair bugs, but it uses a model checking instead of a test suite to evaluate program variants. In summary, there is a module with a model checking that receives the candidate program states and measures a fitness level according to the number and type of the errors found while traversing all model paths. Despite the fitness, the evaluation does not depend on test cases, the model checking requires a set of properties to be verified, for example, built-in properties or custom assertions. Meanwhile, our proposal does not require any predefined properties because our assumption is that they can be captured automatically by the learning model.

It is known that, in practice, a test Oracle may not be available or its usage is too expensive. Thus, [32] applied the concept of Metamorphic Test (MT) aiming to alleviate the test Oracle problem for generating program repair. MT verifies the relations among multiple test cases and their outputs instead of checking individual test cases correctness. Despite that, the proposal does not require an oracle to guide the search process, it is necessary to run a set of MT to evaluate a program variant.

Hence, one alternative for costly test suite-based evaluation approach is the static analysis of "naturalness". The "naturalness" of software was studied in [16], where the authors investigate the assumption that codes of programs are likely to be repetitive and predicted, as the natural languages. Using a common language model called n-grams, it was possible to capture regularity on the tokens in Java and C corpus of code. Based on the previous discovery, it was implemented an Eclipse plugin to improve the code completion engine of the Eclipse IDE. By considering a training corpus of Java projects the new plugin outperformed the native one especially for predicting little tokens. So the n-gram frequency is associated with the "naturalness".

Inspired by great results of language models and the embedding words for text mining, we speculated that these methods can also help the evaluation process in the field of automatic program repair. So we proposed our method based on Word2vec. The Word2vec has been used on many problems, for instance, it was used in [71] to support predictions on Name Entity Recognition problem. That is, given a word, it must be identified as a location, person or organization. The authors report interesting results by combining Word2vec and Linear Support Vector Classification algorithm on the task of classifying words from a corpus of newspaper articles.

Although some works try to evaluate variants with lower cost and greater effectiveness, we did not find any work that used sequence-to-sequence model or embedded words for this.

5.4 Summary

In this chapter, we have shown how to accelerated the generation of word embeddings used in the context of Automated Program Repair. In particular, the word embedding implemented is an extension of the well known Word2Vec with the CBOW model. We exploited fine-grained parallelism by launching thousands of threads of a manycore architecture to process words of textual datasets. In this process we developed a new way of choosing negative samples that is beneficial for exploiting parallelism. This allowed us to efficiently produce the embedding-based metrics needed for Automated Program Repair systems. As additional contributions, we have introduced two word embedding-based metrics useful to differentiate program variants.

CHAPTER 6

Experiments and Results

In this chapter, we present the experiments conducted for the FaSST-*k*NN and sFaSST-*k*NN algorithms which can be seen in Section 6.1 and 6.2 respectively. And we present in Section 6.3 the experiments to evaluate the acceleration of fine-grained parallelism word embedding. Also, we present in Section 6.4 the experiments performed on the application of word embeddings for Automated Software Repair.

6.1 FaSST-*k*NN

FaSST-*k*NN is our fine-grained parallel algorithm, that applies filtering techniques based on the most common important terms of the query document using tf-idf.

6.1.1 Experimental Evaluation

The experimental work was conducted on a machine running Debian 9.4, with a Intel Xeon E5-2620, 16GB of ECC RAM, and four GeForce Nvidia GTX Titan Black, with 6GB of RAM and 2,880 cuda cores each. The CPU code was compiled with GCC 6.3.0 while the GPU code used the compiler provided by the CUDA 9.0. All the codes targeted the native architecture and had the O3 optimization flag set. In order to consider the costs of all data transfers in our experiments, we report the wall times on a dedicated machine so as to rule out external factors, like high load caused by other processes. The reported numbers are average of 10 independent runs.

In order to evaluate the *k*NN search, we consider a large real-world textual dataset, MEDLINE, which has the characteristics of high dimensionality and sparsity. For it, we performed a traditional preprocessing task: we removed stopwords, using the standard SMART list, and applied a simple feature selection by removing terms with low "document frequency (DF)"[1]. Regarding term weighting, we used TF-IDF. The specific

[1] We removed all terms that occur in less than six documents (i.e., DF<6).

details of the resulting MEDLINE are: 268,766 terms, 861,454 documents and 30.88 density[2].

We compare the computation time to perform a *k*NN search using the following algorithms: (1) **FaSST-*k*NN**, (2) **SFaSST-*k*NN**, our GPU-based implementations of *k*NN; (3) **G-KNN**, a GPU *k*NN implementation using CUDA proposed by Rocha et al. [64]. We chose G-KNN because it is the only exact similarity search implementation we have found that deals with high dimensionality and sparsity found in textual datasets like MEDLINE. We adopted $k = \{32, 64, 128, 256, 512\}$ and sample size as $2 \times k$ in all experiments[3] . To measure our proposals' performance, we selected 20% of the dataset as search queries (Around 172,290 documents).

Computational Time

Table 6.1 shows the total time to process all queries using ours and the baseline's *k*NN implementations. From here on, we refer the variations of FaSST-*k*NN as: "NoFilter"[4] for a version that does not use our proposed filtering; "Random" for a random sampling scheme; "M1" for sampling method 1; "M2-T*X*" for sampling method 2, where X indicates the value of its parameter T (the number of terms). We only show $T = \{2, 3, 4\}$ due to space restrictions.

As can be seen, our filtering methods shows a higher efficiency with lower k, going from 441 to below 200 seconds, since less time is spent on sampling. The random sampling had no advantage, showing that our methods works. M2 ended up selecting better samples than M1, having a lower processing time. In this test, the parameter $T = 3$ achieved its best overall performance. We show the speedup regarding M2-T3.

FaSST-*k*NN shows significant speedups over the G-KNN implementation, in comparison, reaching up to $37x$ for $k = 32$. This is mainly due to its implementation that computes the distance to all documents, while ours use an inverted index and the filter technique. Our best method achieved a speedup of $2.4x$ when comparing with NoFilter and $k = 32$, showing that the filtering could greatly compact the distance vector. For higher k the speedup gets lower, since more time is spent on sampling on the CPU.

Table 6.8 shows the time and speedup for SFaSST-*k*NN with 2 and 4 GPUs, comparing NoFilter and M2-T3. The performance of M2-T3 over NoFilter remained similar to when 1 GPU was used, and over M2-T3 with 1 GPU (1-M2-T3) the speedup was around the ideal of $2x$ and $4x$, for 2 and 4 GPUs, respectively. This shows that our GPU *k*NN search scales well with more devices[5].

[2] Density is the average number of terms in a document.
[3] A higher ratio yields a higher compaction ratio, but also a greater sampling time.
[4] This is like GT-*k*NN [11], but with Thrust library functions for sorting and prefix-sum.
[5] An ideal speedup is also expected for G-KNN if it used more GPUs.

Table 6.1: *Query times in seconds and speedups to find the K nearest neighbors with 1 GPU.*

	Query time							M2-T3 Speedup	
K	G-KNN	NoFilter	Random	M1	M2-T2	**M2-T3**	M2-T4	G-KNN	NoFilter
32	6657	441.09	521.71	188.49	180.35	178.88	176.52	37.21	2.47
64	6674	459.55	542.19	216.43	202.66	204.19	208.97	32.68	2.25
128	6694	486.10	581.88	262.78	258.83	256.30	257.48	26.12	1.90
256	6718	522.40	656.69	358.84	343.55	342.72	343.78	19.60	1.52
512	6741	586.62	850.89	523.76	517.08	515.24	522.43	13.08	1.14

Table 6.2: *Query times in seconds and speedups to find the K nearest neighbors with 2 and 4 GPUs.*

	2 GPUs				4 GPUs			
	Query time		Speedup M2-T3		Query time		Speedup M2-T3	
K	NoFilter	M2-T3	NoFilter	1-M2-T3	NoFilter	M2-T3	NoFilter	1-M2-T3
32	229.34	87.69	2.62	2.04	115.50	45.13	2.56	3.96
64	233.18	104.28	2.24	1.96	117.45	51.29	2.29	3.98
128	241.76	126.99	1.90	2.02	121.54	63.32	1.92	4.05
256	258.13	173.07	1.49	1.98	127.38	85.19	1.50	4.02
512	293.43	257.70	1.14	2.00	142.45	127.36	1.12	4.05

Runtime Profiling

We show the impact of sorting the inverted index before doing the sampling method on Table 6.3. For $k = 32$, the compaction ratio up to more than 3 times when using the sorted version, while for $k = 512$ it increases up to 1.6 times. For $k = \{62, 128, 256\}$ this value decreased from $3x$ to $1.6x$ as k increased. This confirms that our proposals benefits further from this sorting of the data. The random sampling achieved only 1.05x of compaction.

The FaSST-*k*NN and SFaSST-*k*NN sampling and sorting times are shown in Tables 6.4 and 6.5. It represents the sum of time spent in these operations in each query. Since the sampling is done on the CPU, we can see that the time spent on it is quite high, specially for $k = 512$, reaching the hundreds of seconds for sampling method 2 (M2). This is partially due to associating a single CPU thread to a GPU, lacking any multicore parallelism or overlap of data transfer and computation. Its time decreases almost linearly with the number of GPUs. It also shows that, despite M2 taking more time in sampling, the total and sorting times decreased enough to make it the better method.

Table 6.5 shows that the sorting time decreases greatly with our proposed methods, going from 405 to just 20 seconds with $k = 32$ and the best method M2-T3. As k goes up to 512, the gain over NoFilter decreases, since the compaction ratio also decreases when using the selected sample size of $2 \times k$.

Although not listed in any table, the CPU I/O, indexing, and inverted index sorting times, are 12, 1 and 1.2 second, respectively. For 1 GPU, the threshold selection time was 1 and 6 seconds for $k = 32$ and $k = 512$, respectively. And the compaction time was 24 and 38 seconds, for these same values of k. With multi-GPUs these times got

Table 6.3: *Impact of the sorted inverted index on the compaction ratio.*

K	Data state	M1	M2-T2	M2-T3	M2-T4
32	unsorted	107.19x	106.42x	90.73x	77.44x
32	sorted	214.78x	278.27x	277.80x	268.98x
512	unsorted	10.09x	41.49x	39.66x	37.29x
512	sorted	16.68x	51.92x	52.92x	53.40x

Table 6.4: *Sum of sampling times in seconds.*

	1 GPU					2 GPUs			4 GPUs		
K	Random	M1	M2-T2	M2-T3	M2-T4	Random	M1	M2-T3	Random	M1	M2-T3
32	6.18	24.42	24.12	25.07	22.34	3.28	11.00	11.36	1.62	5.33	5.75
64	10.36	28.49	27.53	29.51	31.11	5.30	13.65	14.44	2.62	6.48	6.82
128	18.11	34.67	38.96	40.01	39.79	9.41	17.09	19.34	4.63	7.97	9.06
256	31.84	49.29	56.89	58.29	57.85	17.80	24.74	29.90	9.35	11.71	14.17
512	58.11	76.01	96.22	102.19	106.88	35.42	40.68	51.99	19.34	20.04	24.81

Table 6.5: *Sum of sorting times in seconds.*

	1 GPU					2 GPUs			4 GPUs		
K	NoFilter	M1	M2-T2	M2-T3	M2-T4	NoFilter	M1	M2-T3	NoFilter	M1	M2-T3
32	405.73	26.78	21.66	20.57	21.82	200.07	13.87	10.68	107.32	7.09	5.70
64	406.39	35.76	28.47	27.13	28.44	201.20	18.50	14.46	106.62	9.52	7.30
128	405.57	49.36	41.78	39.15	40.21	201.89	26.32	20.11	105.85	14.02	10.63
256	406.13	74.66	59.34	57.62	57.92	202.39	39.34	30.37	103.40	21.52	16.04
512	408.87	112.74	85.96	84.62	86.71	202.72	60.98	44.91	101.20	33.10	24.57

almost ideals speedups. The FaSST-*k*NN performs multiple queries exceptionally well but it excels at single query *k*NN search. Once the dataset has been read, moved to the GPU and indexed, subsequent queries can be processed very fast. Considering 1 GPU and $k = 512$, FaSST-*k*NN takes 3 milliseconds[6] at average to process a single query, making it suitable for on-the-fly top k search using real datasets.

6.2 SFaSST-*k*NN

FaSST-*k*NN is our fine-grained parallel algorithm, that applies filtering techniques based on the most common important terms of the query document using tf-idf and a distributed inverted index in the multi-GPU system.

6.2.1 Experimental Evaluation

The experimental work was conducted on a machine running Debian 9.4, with a Intel Xeon E5-2620, 16GB of ECC RAM, and four GeForce Nvidia GTX Titan Black, with 6GB of RAM and 2,880 cuda cores each. The CPU code was compiled with GCC 6.3.0 while the GPU code used the compiler provided by the CUDA 9.0. All the codes targeted the native architecture and had the O3 optimization flag set. In order to consider the costs of all data transfers in our experiments, we report the wall times on a dedicated

[6] We calculated with the time of method 2 with $T = 3$ and the size of the query set (172,290).

machine so as to rule out external factors, like high load caused by other processes. The reported numbers are average of 10 independent runs.

In order to evaluate the *k*NN search, we consider two large real-world textual datasets, Medline and PubMed, which both have the characteristics of high dimensionality and sparsity. For both, we performed a traditional preprocessing task: we removed stopwords, using the standard SMART list, and applied a simple feature selection by removing terms with low "document frequency (DF)"[7]. Regarding term weighting, we used TF-IDF. The specific details of the resulting datasets are described in Table 6.6, showing number of terms, documents, non-zero entries, density[8], and sparsity[9]. With that, each PubMed entry's size is closer to the real size of a document.

Table 6.6: *General information on the datasets.*

Dataset	# terms	# docs	# non-zeros	Density	Sparsity
Medline	362,717	861,454	26,537,087	30.805	0.0085%
PubMed	958,067	729,937	624,260,424	855.223	0.0892%

We compare the computation time to perform a *k*NN search using the following algorithms: (1) **FaSST-*k*NN**, (2) **SFaSST-*k*NN**, our GPU-based implementations of *k*NN; (3) **G-KNN**, a GPU *k*NN implementation using CUDA proposed by Rocha et al. [64]. We chose G-KNN because it is the only exact similarity search implementation we have found that deals with high dimensionality and sparsity found in textual datasets like Medline and PubMed. We adopted $k = \{32, 64, 128, 256, 512\}$, sample size as $2 \times k$ in all experiments[10], and batch size $B = \{1, 5, 10\}$. To measure our proposals' performance, we selected 20% of the dataset as search queries.

Computational Time

From here on, we refer as "NoFilter"[11] a version of FaSST-*k*NN that does not use our proposed filtering, and "SNoFilter" for a version of SFaSST-*k*NN. When using a single GPU, both implementations produce the same results. Table 6.16 shows the total time to process all queries of Medline using our *k*NN implementations. We only show $B = \{1, 5, 10\}$ due to space restrictions, and also because there were no significant improvements with $B > 10$.

As can be seen, our filtering method can reduce the time to find the k nearest neighbors from 300 to 138 seconds, when using a single GPU in both implementations. For FaSST, the use of batch size 5 is enough to provide a small gain in performance, while

[7] We removed all terms that occur in less than six documents (i.e., DF < 6).
[8] Density is the average number of terms in a document.
[9] Sparsity is the percentage of Density per number of terms.
[10] A higher ratio yields a higher compaction ratio, but also a greater sampling time.
[11] This is like GT-*k*NN [11], but with Thrust library functions for sorting and prefix-sum.

Table 6.7: *Query times in seconds to find the K nearest neighbors in Medline.*

			FaSST					NoFilter				
	B	K	32	64	128	256	512	32	64	128	256	512
1 GPU	1		119.53	121.13	124.35	129.00	137.67	289.40	291.49	292.21	294.39	299.23
	5		111.39	113.49	116.10	121.00	130.16	291.11	291.87	292.50	294.97	299.45
	10		111.26	113.31	116.06	121.50	130.27	292.92	293.63	294.49	296.60	300.75
4 GPUs	1		29.74	30.15	30.85	32.23	34.18	70.87	71.87	72.15	72.69	73.84
	5		27.72	28.08	28.94	30.20	32.27	71.59	72.17	72.24	72.77	73.96
	10		27.88	28.41	29.04	30.21	32.40	72.06	72.43	72.64	73.48	74.53
			SFaSST					SNoFilter				
1 GPU	1		120.19	122.38	124.96	130.19	138.67	290.10	292.25	293.01	295.47	299.97
	5		111.58	113.47	116.63	121.35	130.40	291.60	292.30	292.85	295.41	299.46
	10		111.55	113.56	116.37	121.52	130.37	292.96	293.67	294.53	296.72	301.26
4 GPUs	1		67.16	68.81	72.11	79.47	94.41	102.07	107.40	97.58	98.22	99.41
	5		45.79	47.52	51.24	57.81	72.00	88.43	89.38	90.64	93.09	96.75
	10		44.68	46.17	50.22	56.62	70.29	89.35	90.50	90.56	93.41	97.18

Table 6.8: *Speedups with 4 GPUs and over NoFilter solutions in Medline.*

			FaSST 4 GPUs speedup					NoFilter 4 GPUs speedup				
	B	K	32	64	128	256	512	32	64	128	256	512
	1		4.02	4.02	4.03	4.00	4.03	4.08	4.06	4.05	4.05	4.05
	5		4.02	4.04	4.01	4.01	4.03	4.07	4.04	4.05	4.05	4.05
	10		3.99	3.99	4.00	4.02	4.02	4.07	4.05	4.05	4.04	4.04
			SFaSST 4 GPUs speedup					SNoFilter 4 GPUs speedup				
	1		1.79	1.78	1.73	1.64	1.47	2.84	2.72	3.00	3.01	3.02
	5		2.44	2.39	2.28	2.10	1.81	3.30	3.27	3.23	3.17	3.10
	10		2.50	2.46	2.32	2.15	1.85	3.28	3.24	3.25	3.18	3.10
			FaSST speedup over NoFilter					SFaSST speedup over SNoFilter				
1 GPU	1		2.42	2.41	2.35	2.28	2.17	2.41	2.39	2.34	2.27	2.16
	5		2.61	2.57	2.52	2.44	2.30	2.61	2.58	2.51	2.43	2.30
	10		2.63	2.59	2.54	2.44	2.31	2.63	2.59	2.53	2.44	2.31
4 GPUs	1		2.38	2.38	2.34	2.26	2.16	1.52	1.56	1.35	1.24	1.05
	5		2.58	2.57	2.50	2.41	2.29	1.93	1.88	1.77	1.61	1.34
	10		2.58	2.55	2.50	2.43	2.30	2.00	1.96	1.80	1.65	1.38

NoFilter actually loses a bit. This behavior remains the same with 4 GPUs for both, while decreasing the query time by 4 times, getting as low as 28 seconds. As for SFaSST, due to the query sharing among each CPU thread that controls a GPU, a higher batch size yields better performance since each query can be processed (read and sampling steps) in parallel. As expected, the synchronization between the threads and the merging of the results increased the query time up to 38 seconds more in comparison with FaSST, when $k = 512$. But SFaSST remains the sole version that can deal with datasets bigger than the GPU's memory size.

Table 6.8 shows the speedups of our implementations with 4 GPUs, and over the NoFilter versions. For FaSST and NoFilter with 4 GPUs, due to the independent query processing between GPUs, an ideal speedup was achieved. For SFaSST, the highest speedup is only 2.5x because of the query sharing where other threads need to wait for a single thread, and also the synhcronization and merging steps. This also occurs because the time spent in GPU is really low in comparison to the CPU time. This shows

Table 6.9: *Query times in seconds and speedups to find the K nearest neighbors in PubMed.*

	1 GPU		4 GPUs				Speedup 4 GPUs	
K	SFaSST / FaSST	SNoFilter / NoFilter	FaSST	NoFilter	SFaSST	SNoFilter	FaSST	SFaSST
32	6404.39	6544.51	1597.69	1630.42	975.74	1007.28	4.01	6.56
64	6407.63	6548.60	1601.55	1634.82	977.86	1008.68	4.00	6.55
128	6412.09	6551.27	1607.31	1640.30	981.05	1010.50	3.99	6.54
256	6419.71	6556.00	1612.39	1646.28	985.28	1013.64	3.98	6.52
512	6433.49	6567.51	1619.07	1654.42	991.52	1017.25	3.97	6.49

Table 6.10: *G-KNN comparison with FaSST / SFaSST using 1 GPU and batch size 10.*

		G-KNN time (s)					FaSST speedup				
Dataset	K	32	64	128	256	512	32	64	128	256	512
Medline		6657	6674	6694	6718	6741	59.83	58.90	57.68	55.29	51.74
PubMed		359338	359340	359343	359348	359357	56.11	56.08	56.04	55.98	55.86

in SNoFilter speedup, since the more time spent on sorting the result in GPU ended up giving it a higher speedup. When comparing FaSST with NoFilter, just the reduction of the sorting time gave up to 2.6x higher efficieny, showing how demanding that step is in k nearest neighbor search. The lowest efficiency gain was with SFaSST when using $k = 512$, since higher k also means higher merging time on CPU.

Table 6.9 shows the total query time and speedups for our implementations when using PubMed. Due to the higher density, using $B > 1$ actually increased the overall time so the table only shows batch size $B = 1$. Also due to density, and higher number of non-zeros in the inverted index, the distance calculation phase had a higher impact on the overall time. Thus, the sorting phase where the filter is used to reduce the number of elements to be sorted, ends up contributing just a little for the efficiency. That reduction was 140 seconds at most when $k = 32$, decreasing the total time from 6544 to 6404 seconds. This decrease is similar when using 4 GPUs, with a reduction of around 30 seconds.

For FaSST and NoFilter, its speedups were ideal at 4x, but for the versions with a distributed inverted index a super linear speedup of 6.5x was achieved. Due to the distribution of documents between GPUs, the number of positions for the distances' vector is 4 times lower in SFaSST than FaSST. One implementation detail is that each document's distance is updated atomically in the GPU's highly parallel code. And having less positions to update means a higher chance of different atomic operations being coalesced, or hitting the same cache lines. This was proven by running the NVIDIA profiling tool *nvprof*, where it showed that the atomic throughput for FaSST was 339 GB/s and for SFaSST 210 GB/s, among other metrics with similar behavior.

Table 6.10 shows significant speedups over the G-KNN[12] implementation, in

[12] An ideal speedup is expected for G-KNN if it used more GPUs.

comparison, reaching up to $60x$ for $k = 32$. This is mainly due to its implementation that computes the distance to all documents, while ours use an inverted index and the filter technique. It specially shows when processing PubMed, which takes around 100 hours to complete.

Runtime Profiling

In Table 6.11 we describe the compaction ratio of the proposed filter in both implementations. For single GPU and FaSST with 4 GPUs the ratio is the same, since the inverted index is the same for during sampling. Only for SFaSST with multiple GPUs that the ratio is different due to each thread doing the sampling on its own associated partial index. The overall behavior is that for lower k the compaction is higher. Medline has a higher ratio in comparison to PubMed due to its lower density, which means that its meaningful terms weigh more in relation to the document's size. The ratio for SFaSST is lower due to the use of partial indexes, which reduces the quantity of good candidates during sampling. This ratio was only achieved due to the sorting of the inverted index prior to the queries processing.

The sampling and compaction times for FaSST and SFaSST in Medline are shown in Table 6.12. It represents the sum of time spent in these operations in each query. The values are the average for each batch size (1, 5, 10) followed by its standard deviation[13]. Even though sampling is done on the CPU, it still takes just a few seconds in the overall time, with the higher batch sizes having the lower times. For FaSST with 4 GPUs the time decreases proportionally, but for SFaSST the time for higher k increases more. This was due to lack of independent processing of queries, which makes the CPUs spend more time doing the sampling of the same query, and the use of partial indexes which decreases the amount of possible samples. This increases the chance of having to complete the $2 \times k$ samples with random ones.

For compaction time, it is the same for all k due to the code having to iterate over all distances for flagging each position. Both implementations have the same time with 1 GPU, and with 4 GPUs FaSST has a linear decrease due to its independent query processing, while this lack of processing overlap makes SFaSST only 4 seconds faster than before.

For the sorting time in Medline for all implementations Table 6.13 shows the values averaged by batch size followed by its standard deviations. For the NoFilter versions the value is the same for all k, since the sorting is always the same. When the proposed filtering method is applied, the time is greatly reduced, from 215 to as low as 12.7 seconds. For single GPU FaSST and SFaSSt are equal, while with 4 GPUs SFaSST is

[13] We present it this way due to space restrictions.

Table 6.11: *Compaction ratios for FaSST and SFaSST using 1 and 4 GPUs.*

	1 GPU (Both) and 4 GPUs (FaSST)		4 GPUs (SFaSST)	
K	Med	Pub	Med	Pub
32	254.89	110.46	113.66	62.06
64	170.81	83.73	75.43	45.90
128	113.29	62.04	48.68	33.99
256	75.47	45.90	30.66	25.64
512	48.61	34.00	19.57	19.20

Table 6.12: *Sum of sampling and compaction times in seconds for Medline with FaSST and SFaSST using 1 and 4 GPUs.*

	Sampling time				Compaction time			
	1 GPU		4 GPUs		1 GPU		4 GPUs	
K	FaSST	SFaSST	FaSST	SFaSST	FaSST	SFaSST	FaSST	SFaSST
32	1.50±0.82	1.67±1.06	0.39±0.22	1.63±1.60	16.00±0.23	15.96±0.22	3.90±0.06	11.60±0.36
64	1.56±0.81	1.76±1.07	0.40±0.22	1.86±1.67	16.05±0.22	16.07±0.19	3.92±0.06	11.68±0.30
128	1.70±0.85	1.87±1.09	0.43±0.22	2.32±1.49	16.08±0.21	16.09±0.19	3.93±0.06	11.83±0.22
256	1.97±0.81	2.12±1.10	0.50±0.23	3.67±0.98	16.12±0.18	16.15±0.15	3.94±0.06	11.90±0.20
512	2.55±0.96	2.68±1.18	0.64±0.23	7.73±0.83	16.18±0.15	16.18±0.17	3.95±0.06	11.96±0.17

Table 6.13: *Sum of sorting times in seconds for Medline for all implementations using 1 and 4 GPUs.*

	1 GPU				4 GPUs			
K	FaSST	NoFilter	SFaSST	SNoFilter	FaSST	NoFilter	SFaSST	SNoFilter
32	12.72±0.46	215.20±0.23	12.72±0.41	215.21±0.28	3.07±0.12	52.00±0.53	12.19±0.80	71.34±0.15
64	13.78±0.41	215.41±0.28	13.72±0.46	215.41±0.14	3.30±0.12	52.20±0.13	12.94±0.73	71.39±0.23
128	15.12±0.41	215.49±0.27	15.07±0.40	215.48±0.15	3.62±0.12	52.11±0.18	14.15±0.53	71.15±0.24
256	17.06±0.47	215.56±0.14	17.09±0.38	215.64±0.08	4.06±0.13	52.10±0.19	15.92±0.52	71.27±0.19
512	19.97±0.27	215.87±0.09	19.97±0.30	215.78±0.20	4.71±0.12	51.92±0.49	18.68±0.51	71.16±0.24

Table 6.14: *Sum of sorting times in seconds for Pubmed for all implementations using 1 and 4 GPUs.*

	1 GPU				4 GPUs			
K	FaSST	NoFilter	SFaSST	SNoFilter	FaSST	NoFilter	SFaSST	SNoFilter
32	13.45	169.56	13.33	169.64	3.40	41.99	11.90	57.48
64	14.67	171.68	14.45	171.73	3.70	42.07	12.82	57.65
128	16.42	171.77	15.90	171.60	4.09	41.98	13.97	57.56
256	18.72	171.77	17.98	171.17	4.58	42.12	15.38	57.85
512	21.28	172.17	21.07	171.55	5.27	42.16	17.27	57.87

slower due to lower compaction ratio and also lack of independence between queries that helps FaSST win over it. For PubMed in Table 6.14 the behavior is the same, reinforcing that the filter to reduce sorting time works. And Table 6.15 shows the sum of sampling and compaction times of PubMed, which also has equal behavior to the one described for Medline. It takes just a few seconds in total, even though its a bigger dataset, further showing that in its case the bigger portion of time is spent on distances calculation.

Although not listed in any table, the CPU I/O, indexing, and inverted index sorting times, are 12, 0.3 and 0.15 second, respectively for Medline, and 200, 1.8 and 0.6 second for PubMed. FaSST-*k*NN performs exceptionally well and it excels at *k*NN search. Once the dataset has been read, moved to the GPU and indexed, subsequent queries can be processed very fast. Considering 1 GPU and $k = 512$, FaSST-*k*NN takes

Table 6.15: *Sum of sampling and compaction times in seconds for Pubmed with FaSST and SFaSST using 1 and 4 GPUs.*

	Sampling time				Compaction time			
	1 GPU		4 GPUs		1 GPU		4 GPUs	
K	FaSST	SFaSST	FaSST	SFaSST	FaSST	SFaSST	FaSST	SFaSST
32	2.69	3.14	0.68	4.15	13.30	13.18	3.35	9.89
64	2.80	3.09	0.69	4.18	13.23	13.32	3.37	9.83
128	2.79	3.16	0.73	4.23	13.37	13.15	3.39	9.92
256	3.04	3.49	0.76	4.35	13.72	13.49	3.38	9.76
512	3.38	3.79	0.83	5.01	13.49	13.58	3.44	9.86

0.75 millisecond at average to process a single query of Medline and 26.85 milliseconds of PubMed, making it suitable for on-the-fly top k search using real datasets. And for datasets that do not fit the memory of a single GPU, SFaSST can be used to deal with those while practically having the same efficiency.

6.3 Accelerating word embedding generation with fine-grain parallelism

6.3.1 Experimental Evaluation

In this context, there are two open questions that are going to guide this research:

- How to exploit parallelism in text representation in an efficient and scalable way?
- How to apply the parallel proposals in the context of Automated Program Repair application?

The experimental work was conducted on a machine running Debian 9.4, with 2x Intel Xeon E5-2620, 16GB of ECC RAM, and four GeForce Nvidia GTX Titan Black, with 6GB of RAM and 2,880 CUDA cores each. The CPU code was compiled with GCC 6.3.0 while the GPU code used the compiler provided by the CUDA 9.0.

Corpus and Parameterization

We experimented with the following corpus:

1 Billion Word Language Model Benchmark - A standard corpus used in language modeling [12]. Parameters used for this dataset: window size 10, minimum frequency 5 and random word discard set to 1e-4.

Defects4J - A collection of reproducible bugs and a supporting infrastructure with the goal of advancing software engineering research [36][14]. Parameters used for this dataset: window size 3, minimum frequency 1 and no random word discard.

[14] Available at https://github.com/rjust/defects4j

The remaining parameters for both datasets were: 20 iterations, 10 unigram negative samples and hidden layer size $N \in \{200, 400\}$. When kFN was applied, the initial kFN iteration was 16 and $k \in \{5, 10, 20\}$. Results are an average of 10 runs.

Defects4J Preprocessing

The source codes were processed in the following way: i) Removal of comments and Java annotations; ii) Space inserted at the sides of each non-alphanumeric character (i.e, like operators and brackets); iii) Literals substituted by corresponding tokens (INT_TOKEN, STRING_TOKEN, and so on); iv) Compound operators like "+=" are rejoined after step ii. The remaining steps are for variables and variable types substitution. The new variable name is built according to each element that precedes it. Modifiers like final, static, private, $< T >$, [ ], etc are mapped to a token of its own and then are appended to the new name. Primitive types like int or float are left as is, while classes are given a new name considering the amount of nested generics ($< T >$) that it has. Finally, the variable type, whether it is a class or not, is appended to the variable name. This is first applied to method's declaration variables, then for every variable declaration succeeded by the "=" operator. Finally, for all words preceding an assignment or comparison statement, which has not been substituted, a generic token is used to replace it. This tokenization process is not perfect, due to difficulty in finding regular expressions that match the variables exactly. Also, variables with the same name but in different scopes, will have the same new name since substitution is done in the whole file.

Experiments Discussion

Our first experiment was to evaluate the performance of our algorithm with kFN used to select the negative samples against the Mikolov implementation. Also, we tested our parallel implementation of Word2Vec without the use of kFN. In the Table 6.16, we tested our algorithm using 1 or 4 GPUs[15]. The hidden layer size of the Word2Vec neural network was defined with 200 and 400 neurons. When the hidden layer of the Word2Vec neural network has 200 neurons, our non-kFN algorithm can match the performance of our baseline when scaling a GPU to 4 GPUs. Using kFN our algorithm performs close to our baseline. Although kFN has not improved the accuracy of Word2Vec when comparing to Mikolov's, as can be seen in the Table 6.17. A smaller accuracy was expected due to the highly parallel GPU and the synchronization step, but the use of kFN actually improved over the GPU version without kFN, the highest being with $k = 20$. In addition, kFN makes it possible to apply parallel algorithms that have excellent performance when working

[15] In our testing environment, we have no more than 4 GPUs for the experiment.

with dense vectors. This opportunity is interesting for algorithms that do not use artificial neural networks to generate word vectors, such as using Pointwise Mutual Information Matrix (PMI) [46]. Our algorithm has linear speed-up relative to our baseline when we use 4 GPUs as can be seen in the Table 6.18 with $N = 400$ having higher speedup due to more exploitation of data parallelism in the GPU.

6.3.2 Results

Table 6.16: *Execution time for hidden layer size 200 and 400.*

$N = 200$									
	No kFN			k = 5		k = 10		k = 20	
	Mikolov	**1 GPU**	**4 GPUs**	**1 GPU**	**4 GPUs**	**1 GPU**	**4 GPUs**	**1 GPU**	**4 GPUs**
1 Billion	84043	34934	8536	37645	9199	38827	9508	41776	10249
Defects4J	25212	20617	5233	20130	5122	20102	5081	21885	5573
$N = 400$									
1 Billion	140437	42567	11015	43666	10454	46247	11219	48624	11770
Defects4J	51609	23944	5819	23551	5800	23997	5764	25562	6280

Table 6.17: *Accuracy percentage for 1Billion with hidden layer size 200 and 400.*

	No kFN			k = 5		k = 10		k = 20	
	Mikolov	**1 GPU**	**4 GPUs**	**1 GPU**	**4 GPUs**	**1 GPU**	**4 GPUs**	**1 GPU**	**4 GPUs**
$N = 200$	64.71	62.89	62.38	61.98	63.57	61.77	63.06	61.65	62.52
$N = 400$	58.10	62.62	63.28	62.94	64.16	62.26	63.92	61.86	64.16

6.4 A new word embedding approach to evaluate potential fixes for APR

We present in this section the experiments and results of each program regarding metrics *Prob* and *Dist*.

Table 6.18: *Speedup for 4 GPUs x Mikolov*

	$N = 200$			
	No kFN	k = 5	k = 10	k = 20
1 Billion	9.84	9.13	8.83	8.2
Defects4J	4.82	4.92	4.96	4.52
	$N = 400$			
1 Billion	12.75	13.43	12.51	11.93
Defects4J	8.87	8.9	8.93	8.22

6.4.1 Experiments

Setup

The experiments were conducted on a machine running CentOS 7.2 64-bits, with an Intel® Xeon®E5-2620 2GHz and 16GB RAM. Our proposal is as follows:

- **Phase 1 (Corpus Selection)**: we use the programs checksum, digits, grade, median, smallest and syllables from IntroClass[16] to validate our proposal. We create a corpus composed of all IntroClass benchmark codes (fixes made by humans). The idea is to capture the naturalness of code from the programming community in general.
- **Phase 2 (Tokenization):** we split each fixed code in a corpus line-by-line. Thus, a token is an entire line because we speculate that the line has enough semantic.
- **Phase 3 (Learning):** we use a token as context and the next one as a target. Then, each pair (context, target_token) in a training fixed code is a sample to train the model. We use the Word2vec toolkit to fit the model's parameters to recognize the sequence. Each token has a representative vector. The vector size produced in most WordVec applications is between 100 and 500 units (neurons). These applications usually have vocabularies with hundreds of thousands of words. Since the vocabulary used in programming is usually small, we have chosen to reduce the word vector size to 50. The chosen Word2Vec architecture was CBOW because this model predicts which token is more likely to happen given another context token. Also, CBOW is faster to train and it has better accuracy for more frequent words than Skip-gram architecture. In addition, Skip-gram model works best for small databases and when we need to represent rare words or phrases[17]. The size of the context window used in model training was one (1). This size captures pieces of codes that have a more recurring neighborhood. Therefore, we can score codes that have more naturalness. This phenomenon was observed empirically in several tests performed with many windows.
- **Phase 4 (Evaluation)** we use two metrics based on Word2Vec softmax output and embedded words distances. These are explained in section 6.4.1.

Metrics

We use two metrics *Prob* and *Dist* based on the proposal's Phase 4 outputs. The metric *Prob* uses the pairwise probability obtained from the last layer of Word2vec. Thus, for each variant evaluated, the metric *Prob* is the average of the pairwise probabilities

[16] https://github.com/ProgramRepair/IntroClass
[17] https://code.google.com/archive/p/word2vec

(context, target_token) obtained from it. Let M denote the number of tokens in a fix, the metric *Prob* is calculated by (6-1).

$$Prob(variant) = \frac{\sum_{i=1}^{M-1} probability(token_i, token_{i+1})}{M} \quad (6\text{-}1)$$

The metric *Dist* uses the vectors, semantically meaningful representations, obtained from the embedded word layer of Word2vec. The metric *Dist* in this work is the Word Mover's Distance (WMD). The WMD distance measures the dissimilarity between two documents as the minimum amount of distance that the embedded words of one document need to "travel" to reach the embedded words of another document [40]. Since this metric evaluates the dissimilarity, we choose the fixed code in the corpus with the highest *Prob* as the reference for comparison with the variant under evaluation.

Scenarios

We evaluated the proposal on three scenarios. For each scenario, we simulate variants applying different mutation operators. We applied the operator one to ten times in a reference code (highest *Prob* on corpus) to analyze the impact of naturalness reduction on evaluation. Thus, we analyzed whether the metrics are sensitive to changes.

The first scenario applies the delete operator to generate variants; the second one applies the insert operator; and the third applies the swap operator.

All operators were applied on different levels, and the result for each level is the average of ten independent applications of the operator. Thus, the level one is the single random application of an operator, the second level is the random application of an operator two times and so on. Consequently, each level represents a reduction level of naturalness, since more perturbation and entropy in the code causes less naturalness.

Therefore, we aim to answer the following research questions with the scenarios aforementioned:

- **RQ1)** Is the metric *Prob* able to capture the loss of naturalness of variants?
- **RQ2)** Is the metric *Dist* able to capture the loss of naturalness of variants?

6.4.2 Results

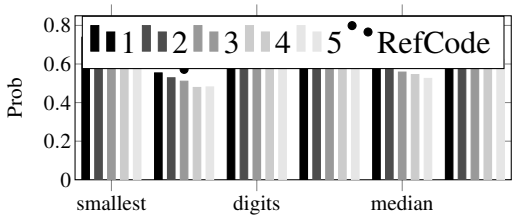
Overall, as the perturbation level increases, we observe an increase in the distance and a decrease in the probability. This behavior can be more easily seen in Figures 6.1(a), 6.2(a) and 6.3 for the metric *Prob*. For the metric *Dist* on Figures 6.1(b) and 6.2(b). The metric *Dist* for the Swap operator is always 0 since the order of the tokens (line of code) does not affect it. Thus the variant is equal to the reference fix for this metric.

The results show that the variants' *Prob* decreased when lines of RefCode were deleted. In some cases, the probability score can increase when the level goes up. For example, the metric *Prob* increased at level 5 for checksum and digits, and at levels 3 and 4 for grade. This happens when a line of code that occurs less frequently in the corpus is removed. Thus, we note that it is possible for an incomplete variant to have a higher score than a fix, although this occurs rarely.

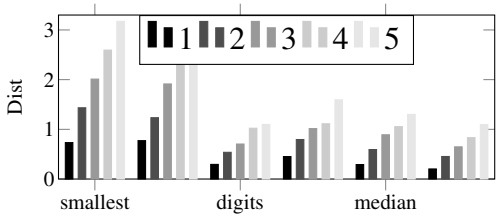
However, the metric *Dist* does not present exceptions. As the perturbation level increases, the metric *Dist* also increases or at least maintains equal to the previous level.

The insert operator causes the variant score to decrease as the level increases since equal tokens out of sequence are not expected. As for the delete operator, tokens from the training corpus of the model that have low frequency can be removed. This can cause the score to diminish with less intensity or even rise. Therefore, the insert operation had more linear behavior than the delete operation.

When the metric *Prob* results in a high score for a variant, we can use the metric *Dist* to see how far the variants are from the RefCode, as shown in the charts 6.1(b) and 6.2(b). The increase in mutation level results in a higher distance from it in relation to the RefCode.

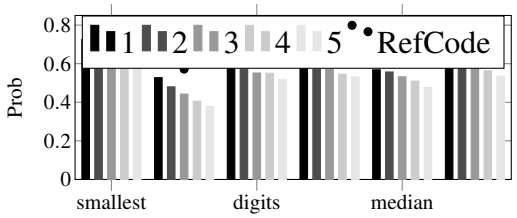


((a))

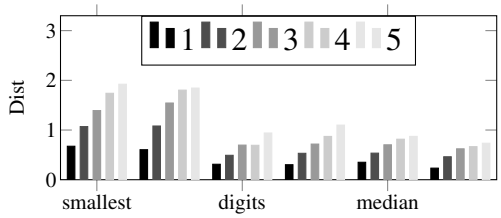


((b))

Figure 6.1: *Results after Delete operator: (a) Prob (b) Dist*



((a))



((b))

Figure 6.2: *Results after Insert operator: (a) Prob (b) Dist*

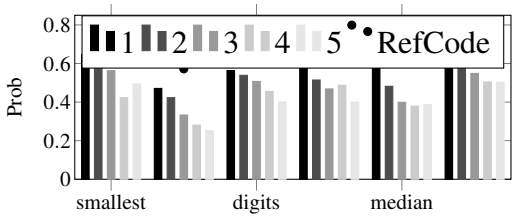


Figure 6.3: *Prob results after Swap operator*

There were benchmark programs that did not behave as expected in terms of decreasing the metric *Prob* as the level of mutation increases. This is because there are an increased variety of fixes to the same problem (different developers). For instance, the smallest program has the fixes with a strong similar structure whereas the grade program is not.

Metrics based on test cases (baseline) have as a disability the generation of a lot of plateaus. Our metrics have a low probability of generating plateaus. Exceptions occur rarely in the delete operation when the same line of the variant is removed in level 1 as can be seen in Table 6.19 for variants 9 and 10 in the smallest problem and for variants 8 and 9 in the checksum problem. Another exception is the swap operation for the metric *Dist*. Most problems in delete and insertion operations do not occur on a plateau. An example can be seen in the level 1 of the delete operator for the grade problem.

For most cases, the metric *Prob* have good results, as can be seen on charts 6.1(a), 6.2(a) and 6.3, capturing the loss of naturalness of variants. The metric *Dist* is able to capture the loss of naturalness of variants because it obtained linear behavior with the increase of the level of mutation for all problems.

Table 6.19: *Scores of different variants for each program generated by delete operator level 1*

	smallest		checksum		grade	
V. ID	**Prob**	**Dist**	**Prob**	**Dist**	**Prob**	**Dist**
1	0.7162	0.7891	0.5169	0.7271	0.6065	0.443
2	0.7229	0.7013	0.5784	0.6353	0.5847	0.3387
3	0.7690	0.6584	0.5836	0.6489	0.6314	0.38
4	0.7632	0.7891	0.5307	1.0395	0.6244	0.3933
5	0.7142	0.6524	0.5504	0.8576	0.5713	0.5094
6	0.7632	0.6772	0.5817	0.6259	0.5847	0.3387
7	0.7311	0.8467	0.5178	0.6546	0.5943	0.8573
8	0.7690	0.7013	0.5712	0.9185	0.6637	0.4267
9	0.7196	0.729	0.5712	0.9185	0.6980	0.3818
10	0.7196	0.7843	0.5367	0.6967	0.6659	0.4525

In addition to these experiments, we run GenProg, an automated software repair method, to generate smallest problem fixes called variant A and variant B as can be seen in Figure 6.4. The purpose of this experiment is to verify whether *Prob* and *Dist* metrics can distinguish variants that have alike or equal fitness values measured only by test case coverage. The metric *Dist* shows how semantically far the candidate to fix is from a correct program that was selected from known fixes.

In a qualitative analysis of the codes, we notice the variant B is closer to the correct program for smallest since the variant A has the statement *return* within blocks of if-clauses. It is not common for this statement to be present within conditional blocks in the C language, thus, the metric *Prob* was able to capture that the co-occurrence between the neighboring statements of the statement *return (0)* is not common.

Whereas the *Prob* value for variant B is 0.74, indicating that it is more likely to be a fix, being each of its pair of subsequent lines more similar to the sequences in the correct code (Figure 6.4). Regarding the *Dist* values, the variant B presents a lower value than variant A. This is natural due to the later has a programming structure closer to the correct program.

Therefore, we can argue the proposal is able to deal with fitness plateaus, that is a huge problem for some automated software repair methods, like GenProg.

Thus, we can answer the research questions:

- **RQ1)** Yes, the metric *Prob* can capture the loss of naturalness of variants, since for most of the cases this metric decreased as the perturbation level increases. Finally, the metric helps to avoid the plateaus, since in only two cases, the generated variants

obtained the same score.

- **RQ2)** Yes, the metric *Dist* is also able to capture the loss of naturalness of variants, since almost all cases this metric increased as the perturbation level increased. The exception was for the swap operator because the tokens remain the same in the variant after the operation.

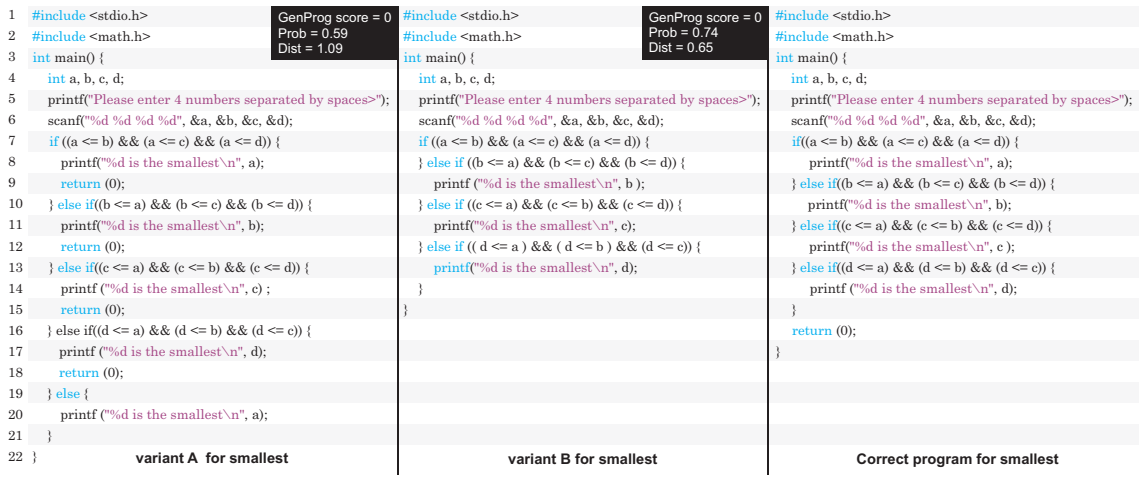


Figure 6.4: *Examples of two canditate variants obtained from GenProg and one correct code for smallest problem.*

6.5 Summary

In this chapter, we present our FaSST-*k*NN and sFaSST-*k*NN experiments for high-dimensionality and sparse text datasets. These algorithms implemented on a GPU improved the top k nearest neighbors search by up to 60x compared to a baseline. Regarding word embeddings experiments, the algorithm implemented on a multi-GPU system scaled linearly and was able to generate embeddings 13x faster than the original multicore Word2Vec algorithm while keeping the accuracy of the results at the same level as those produced by standard word embedding programs. Besides we were able to show that the negative sampling proposed keeps the accuracy of the results at the same level as those produced by standard word embedding programs. The proposed implementation can deal with large corpus, in a computationally efficient way, being a promising alternative to the processing of million source code files needed in Automated Program Repair. Also, we propose new word embeddings-based metrics for evaluating potential fixes in Automated Program Repair.

CHAPTER 7

Conclusions

Intensive use of *k*NN, combined with the high dimensionality and sparsity of textual data, makes it a challenging computational task. We have presented a fine-grained parallel algorithm and a very fast and scalable GPU-based approach for computing the top k nearest neighbors search in textual datasets. Different from other GPU-based *k*NN implementations, we avoid comparing the query document with all training documents. Instead, we use an inverted index in the GPU that is used to quickly find the documents sharing terms with the query document. Although the index does not allow a regular and predictable access to the data, a load balancing strategy is used to evenly distributed the computation among thousand threads in the GPU. Furthermore, we proposed a filtering technique that decreases the sorting time of *k*NN candidates. Our filtering method allows the removal of documents with similarity less than a threshold so that our algorithm spends less time in the sorting phase of the nearest k documents. Our proposal was extended to exploit multi-GPU platforms thus permitting the processing of multiple queries in parallel. We tested our approach in a very memory-demanding and time-consuming task which requires intensive and recurrent execution of *k*NN. Our results show very significant gains in speedup when compared to our baselines.

Given the great advantage of working with complex semantic relationships between words, word embeddings continue to being used in a number of applications. In this work we accelerated the generation of word embeddings used in the context of Automated Program Repair. In particular, the word embedding implemented is an extension of the well known Word2Vec with the CBOW model. We exploited fine-grained parallelism by launching thousands of threads of a manycore architecture to process words of textual datasets. In this process we developed a new way of choosing negative samples that is beneficial for exploiting parallelism. This allowed us to efficiently produce the embedding-based metrics needed for Automated Program Repair systems. In order to validate our proposal we made extensive experimental evaluation using a standard natural language dataset and a novel source code modified dataset. The results showed that our implementation achieves linear scalability on a multi-GPU machine and competitive speedups of up to 13x when compared to the original Word2Vec implementation. In

addition we were able to show that the negative sampling proposed keeps the accuracy of the results at the same level as those produced by standard word embedding programs. The proposed implementation is able to deal with large corpus, in a computationally efficient way, being a promising alternative to the processing of million source code files needed in Automated Program Repair.

Assessing fixes quality generated by Automated Program Repair methods is challenging mainly due to the lack of compliance information, the cost of the task and the number of variables. Several techniques have been applied to evaluate the quality of a fix. The most used is based on test suite execution, especially by search-based program repair methods. But it is costly and imprecise since it generates plateaus, decreasing the search performance. Thus, we proposed a method based on word embeddings to analyze statically the generated variants as to its naturalness and consequently evaluate its quality. To validate our proposal, we used a corpus of fixes from repositories of IntroClass Benchmark. We simulated the variants changing a RefCode in different ways. For each variant, two scores were assigned: *Prob* and *Dist*. Fixes with higher *Prob* and lower *Dist* are considered more natural. Overall, our experiments show promising results, endorsing our hypothesis that embedded words and softmax output from Word2vec can be used as metrics of quality of variants. Therefore, in this context, our main contribution is a method to evaluate statically the variants generated by automated program repair methods. Calculating the metrics for a variant may require more computation as several variants and larger codes need to be evaluated.

7.1 Summary of contributions

With respect to automated program repair application and word embeddings generation, the main contributions of this research can be summarized as:

- A method to statically evaluate variants generated by automated program repair methods;
- A methodology based on embedded words and sequence-to-sequence models to evaluate the naturalness of a potential fix;
- An experimental study using standard program repair benchmarks.
- A multi-GPU implementation that generates word embeddings and achieves linear speed-up on a multi-GPU machine while maintaining the accuracy of the results;
- Development of a new approach for selecting negative samples in the CBOW model;
- Development of a specialized tokenization for a large dataset used in the context of Automated Program Repair;

- Extensive experimental evaluation of the proposed implementation in both texts from natural language and source codes from programming languages.

With respect to *k*NN, the main contributions of this research can be summarized as:

- A threshold-based filtering technique that improves the sorting time of *k*NN candidates;
- A scalable multi-GPU implementation that exploits both data parallelism and task parallelism;
- A distributed inverted index implementation in MultiGPU system;
- The possibility of batch processing multiple queries;
- Extensive experimental work with a standard real-world textual datasets: PubMed.

7.1.1 Discussion and Limitations of our proposals

The FaSST-*k*NN is a very fast GPU-based tool for computing the top *k* nearest in high dimensional and sparse data. However, it has some limitations. First, it is tailored to textual data that can be efficiently represented using an inverted index. It also requires a pre-processing of the dataset to conform to the input format. Second, the tool can only be ran in machines with NVIDIA accelerators, since it has been developed using CUDA. Third, although we were able to process large datasets, the GPU memory limits the maximum size to only a few tens of gigabytes. Fourth, the inverted index is not distributed over the GPU's memory but replicated in each memory, which further increases the memory problem. Finally, it can process similarity search real fast due to its filtering scheme, specially for lower *k* values, and batching of queries. The memory problem was solved by the proposed SFaSST-*k*NN, where the inverted index is distributed among multiples GPUs, turning the solution scalable according to the number of devices. When processing a higher density dataset, PubMed, both solutions could not gain much better efficiency with the filtering scheme, since distance calculation phase had a greater weight in the total query time.

Thus, we propose a new implementation of word embedding generation, in particular, the CBOW architecture (not skip-gram architecture), in manycore architecture that allows increasing the scalability of this algorithm with the insertion of more GPUs.

7.1.2 Published papers

1. A Fast Similarity Search *k*NN for Textual Datasets. In: 2018 Symposium on High Performance Computing Systems (WSCAD), São Paulo, Brazil, 2018, pp. 229-236.(Qualis B3), Publisher: IEEE[1];

2. A Fast Word2Vec implementation on manycore architectures for Text Representation and its applications. In: VI Escola Regional de Informática de Goiás, Goiânia - GO, 14 e 15 de Setembro de 2018 (short paper);
3. A New Word Embedding Approach to Evaluate Potential Fixes for Automated Program Repair. In: 2018 International Joint Conference on Neural Networks (IJCNN) 2018 International Joint Conference on Neural Networks (IJCNN), Rio de Janeiro, 2018, pp. 1-8. (Qualis A1), Publisher: IEEE[4];
4. Accelerating word embedding generation with fine-grain parallelism. In: 8th Brazilian Conference on Intelligent Systems (BRACIS) and the XV Encontro Nacional de Inteligência Artificial e Computacional (ENIAC) (Qualis B2), Publisher: IEEE, 2019;

7.1.3 Manuscripts in review process

1. A Scalable Fast Similarity Search *k*NN for Textual Datasets. Submitted in: Concurrency and Computation: Practice and Experience. (Qualis A2).

7.1.4 Main award

1. Artigo com menção honrosa no XIX Simpósio em Sistemas Computacionais (WSCAD-SSC) - "A Fast Similarity Search *k*NN for Textual Datasets", Sociedade Brasileira de Computação.

7.1.5 Future works

With regard to FaSST-*k*NN, a better performance could be achieved if a filtering scheme was done during the distance calculation, with the use of the Cauchy-Schwarz inequality, for example. Since this work uses weighted vectors and the cosine similarity function to calculate the distance between vectors (documents) of the dataset, Cauchy-Schwarz inequality can be considered to obtain an upper bound pruning limit through partial scalar product estimates between vectors, and thereby reduce the size of the inverted index, eliminate most possible candidates, and decrease the number of full scalar product calculations between dataset vectors.

With respect to automated program repair application and word embeddings generation, to gain more performance and scalability of Word2Vec in many-core architectures, we speculate that the following experiments can accomplish promising results:

- Regarding implementation in Multi-GPU, we can develop new schemes to distribute the gradient for Artificial Neural Network used by Word2Vec. These schemes can be inspired by articles of Deep Learning in Distributed Systems [15];

- Search for a data structure that better exploits the features of manycore architectures (that allows exploiting more parallelism i.e without leaving idle processors) in order to replace the Huffman tree in the Hierarchical Softmax;
- Group words that are most likely to co-occur in order to generate vectors for word classes [69]. As a consequence, the number of vectors that need to be updated will decrease. It is a similar effect when reducing matrices using SVD (Singular Value Decomposition). This idea explores the use of K-Nearest Neighbors algorithm and uniform sampling to approximate the softmax function and achieve $\mathcal{O}(\sqrt{V})$ runtime[1];
- Explore matrix batch multiplication operations for CBOW Architecture in Multi-GPU;
- One way that can increase scalability for word embeddings generation is to replace the use of artificial neural networks with another method of generating co-occurrence matrices. One possibility is to work with word embeddings generation from the PPMI (Positive Pointwise mutual information) matrix. The PPMI matrix generates a very sparse matrix, which makes it possible to apply our SFaSST-*k*N algorithm to speed up the search for vectors that can represent more similar words, tokens or documents.

[1] http://cs231n.stanford.edu/reports/2017/pdfs/130.pdf

References

[1] AFONSO AMORIM, L.; FREITAS, M. F.; DA SILVA, P. H.; MARTINS, W. S. **A fast similarity search knn for textual datasets**. In: *2018 Symposium on High Performance Computing Systems (WSCAD)*, p. 229–236, Oct 2018.

[2] ALEWIWI, M.; ÖRENCIK, C.; SAVAS, E. **Efficient top-k similarity document search utilizing distributed file systems and cosine similarity**. *Cluster Computing*, 2016.

[3] ALLAMANIS, M.; BARR, E. T.; BIRD, C.; SUTTON, C. **Suggesting accurate method and class names**. In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, p. 38–49, New York, NY, USA, 2015. ACM.

[4] AMORIM, L. A.; E FREITAS, M. F.; DANTAS, A.; DE SOUZA, E. F.; CAMILO-JUNIOR, C. G.; MARTINS, W. S. **A new word embedding approach to evaluate potential fixes for automated program repair**. *2018 International Joint Conference on Neural Networks (IJCNN)*, p. 1–8, 2018.

[5] ARCURI, A.; YAO, X. **A novel co-evolutionary approach to automatic software bug fixing**. In: *2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence)*, p. 162–168, June 2008.

[6] ARCURI, A. **Evolutionary repair of faulty software**. *Applied Soft Computing*, 11(4):3494 – 3514, 2011.

[7] ARCURI, A.; YAO, X. **A novel co-evolutionary approach to automatic software bug fixing**. In: *Evolutionary Computation, 2008. CEC 2008.(IEEE World Congress on Computational Intelligence). IEEE Congress on*, p. 162–168. IEEE, 2008.

[8] BELL, N.; HOBEROCK, J. **Thrust: A productivity-oriented library for cuda**. In: *GPU computing gems Jade edition*, p. 359–371. Elsevier, 2012.

[9] BENNETT, K. H.; RAJLICH, V. T. **Software maintenance and evolution: A roadmap**. In: *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, p. 73–87, New York, NY, USA, 2000. ACM.

[10] BLEI, D. M.; NG, A. Y.; JORDAN, M. I. **Latent dirichlet allocation**. *J. Mach. Learn. Res.*, 3:993–1022, Mar. 2003.

[11] CANUTO, S.; GONÇALVES, M.; SANTOS, W.; ROSA, T.; MARTINS, W. **An efficient and scalable metafeature-based document classification approach based on massively parallel computing**. In: *SIGIR*. ACM, 2015.

[12] CHELBA, C.; MIKOLOV, T.; SCHUSTER, M.; GE, Q.; BRANTS, T.; KOEHN, P. **One billion word benchmark for measuring progress in statistical language modeling**. *CoRR*, abs/1312.3005, 2013.

[13] CHEN, D.-K.; SU, H.-M.; YEW, P.-C. **The impact of synchronization and granularity on parallel systems**. *SIGARCH Comput. Archit. News*, 18(2SI):239–248, May 1990.

[14] CHEN, G.; DING, Y.; SHEN, X. **Sweet knn: An efficient knn on gpu through reconciliation between redundancy removal and regularity**. In: *ICDE*. IEEE, 2017.

[15] DEAN, J.; CORRADO, G. S.; MONGA, R.; CHEN, K.; DEVIN, M.; LE, Q. V.; MAO, M. Z.; RANZATO, M.; SENIOR, A.; TUCKER, P.; YANG, K.; NG, A. Y. **Large scale distributed deep networks**. In: *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, p. 1223–1231, USA, 2012. Curran Associates Inc.

[16] DEVANBU, P. **On the naturalness of software**. In: *Proceedings of the 6th India Software Engineering Conference*, ISEC '13, p. 61–61, New York, NY, USA, 2013. ACM.

[17] FORREST, S.; NGUYEN, T.; WEIMER, W.; LE GOUES, C. **A genetic programming approach to automated software repair**. In: *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, GECCO '09, p. 947–954, New York, NY, USA, 2009. ACM.

[18] FRIEDMAN, J. H.; BENTLEY, J. L.; FINKEL, R. A. **An algorithm for finding best matches in logarithmic expected time**. *In TOMS. ACM*, 1977.

[19] GARCIA, V.; DEBREUVE, E.; BARLAUD, M. **Fast k nearest neighbor search using gpu**. In: *CVPR Workshops*, p. 1–6, 2008.

[20] GIESEKE, F.; HEINERMANN, J.; OANCEA, C.; IGEL, C. **Buffer kd trees: processing massive nearest neighbor queries on gpus**. In: *ICML*, 2014.

[21] GOPINATH, D.; MALIK, M. Z.; KHURSHID, S. **Specification-Based Program Repair Using SAT**, p. 173–188. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[22] GOUES, C. L.; NGUYEN, T.; FORREST, S.; WEIMER, W. **Genprog: A generic method for automatic software repair**. *IEEE Transactions on Software Engineering*, 38(1):54–72, Jan 2012.

[23] GRAVE, E.; JOULIN, A.; CISSÉ, M.; GRANGIER, D.; JÉGOU, H. **Efficient softmax approximation for gpus**. *CoRR*, abs/1609.04309, 2016.

[24] GUPTA, R.; PAL, S.; KANADE, A.; SHEVADE, S. **Deepfix: Fixing common c language errors by deep learning**. In: *Proceedings of the 31ft AAAI Conference on Artificial Intelligence*, 2017.

[25] GUPTA, S.; KHARE, V. **Blazingtext: Scaling and accelerating word2vec using multiple gpus**. In: *Proceedings of the Machine Learning on HPC Environments*, MLHPC'17, p. 6:1–6:5, New York, NY, USA, 2017. ACM.

[26] GUTIÉRREZ, P. D.; LASTRA, M.; BACARDIT, J.; BENÍTEZ, J. M.; HERRERA, F. **Gpu-sme-knn: Scalable and memory efficient knn and lazy learning using gpus**. *Information Sciences*, 373:165–182, 2016.

[27] HARRIS, Z. **Distributional structure**. *Word*, 10(23):146–162, 1954.

[28] HUANG, G.; GUO, C.; KUSNER, M. J.; SUN, Y.; SHA, F.; WEINBERGER, K. Q. **Supervised word mover's distance**. In: Lee, D. D.; Sugiyama, M.; Luxburg, U. V.; Guyon, I.; Garnett, R., editors, *Advances in Neural Information Processing Systems 29*, p. 4862–4870. Curran Associates, Inc., 2016.

[29] HWANG, K. **Advanced Computer Architecture: Parallelism,Scalability,Programmability**. McGraw-Hill Higher Education, 1st edition, 1992.

[30] JI, S.; SATISH, N.; LI, S.; DUBEY, P. **Parallelizing word2vec in shared and distributed memory**. apr 2016.

[31] JI, T.; CHEN, L.; MAO, X.; YI, X. **Automated program repair by using similar code containing fix ingredients**. In: *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, p. 197–202, June 2016.

[32] JIANG, M.; CHEN, T. Y.; KUO, F.-C.; TOWEY, D.; DING, Z. **A metamorphic testing approach for supporting program repair without the need for a test oracle**. *Journal of Systems and Software*, 126:127 – 140, 2017.

[33] JIN, J.; CHEN, Q. **A trust-based top-k recommender system using social tagging network**. In: *2012 9th International Conference on Fuzzy Systems and Knowledge Discovery*, p. 1270–1274, May 2012.

[34] JOBSTMANN, B.; GRIESMAYER, A.; BLOEM, R. **Program Repair as a Game**, p. 226–238. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[35] JOHNSON, J.; DOUZE, M.; JÉGOU, H. **Billion-scale similarity search with gpus**. *arXiv preprint arXiv:1702.08734*, 2017.

[36] JUST, R.; JALALI, D.; ERNST, M. D. **Defects4j: A database of existing faults to enable controlled testing studies for java programs**. In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, p. 437–440, New York, NY, USA, 2014. ACM.

[37] KE, Y.; STOLEE, K. T.; GOUES, C. L.; BRUN, Y. **Repairing programs with semantic code search (t)**. In: *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ASE '15, p. 295–306, Washington, DC, USA, 2015. IEEE Computer Society.

[38] KOIKE, A.; SADAKANE, K. **A novel computational model for gpus with applications to efficient algorithms**. *International Journal of Networking and Computing*, 5(1):26–60, 2015.

[39] KUANG, Q.; ZHAO, L. **L.: A practical gpu based knn algorithm**. In: *ISCSCT)*, 2009.

[40] KUSNER, M.; SUN, Y.; KOLKIN, N.; WEINBERGER, K. Q. **From word embeddings to document distances**. In: Blei, D.; Bach, F., editors, *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, p. 957–966. JMLR Workshop and Conference Proceedings, 2015.

[41] KWIATKOWSKI, J. **Evaluation of parallel programs by measurement of its granularity**. In: Wyrzykowski, R.; Dongarra, J.; Paprzycki, M.; Waśniewski, J., editors, *Parallel Processing and Applied Mathematics*, p. 145–153, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[42] LE, Q. V.; MIKOLOV, T. **Distributed representations of sentences and documents**. *CoRR*, abs/1405.4053, 2014.

[43] LE GOUES, C.; DEWEY-VOGT, M.; FORREST, S.; WEIMER, W. **A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each**. In: *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, p. 3–13, Piscataway, NJ, USA, 2012. IEEE Press.

[44] LE GOUES, C.; HOLTSCHULTE, N.; SMITH, E. K.; BRUN, Y.; DEVANBU, P.; FORREST, S.; WEIMER, W. **The ManyBugs and IntroClass benchmarks for automated repair of C programs**. *IEEE Transactions on Software Engineering (TSE)*, 41(12):1236–1256, December 2015. http://dx.doi.org/10.1109/TSE.2015.2454513 DOI: 10.1109/TSE.2015.2454513.

[45] LEVY, O.; GOLDBERG, Y. **Linguistic regularities in sparse and explicit word representations**. In: *Proceedings of the Eighteenth Conference on Computational Natural Language Learning*, p. 171–180, Ann Arbor, Michigan, June 2014. Association for Computational Linguistics.

[46] LEVY, O.; GOLDBERG, Y. **Neural word embedding as implicit matrix factorization**. In: *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'14, p. 2177–2185, Cambridge, MA, USA, 2014. MIT Press.

[47] LIANG, S.; WANG, C.; LIU, Y.; JIAN, L. **Cuknn: A parallel implementation of k-nearest neighbor on cuda-enabled gpu**. In: *IEEE YC-ICT'09.*, 2009.

[48] LIU, R. **Unpublished work, retrieved from the author's github web page https://libraries.io/github/fengchenhpc on 2018-08-15**. 2014.

[49] LONG, F.; RINARD, M. **Automatic patch generation by learning correct code**. In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, p. 298–312, New York, NY, USA, 2016. ACM.

[50] MANNING, C. D.; RAGHAVAN, P.; SCHÜTZE, H. **Introduction to Information Retrieval**. Cambridge University Press, New York, NY, USA, 2008.

[51] MANNING, C. D.; SCHÜTZE, H. **Foundations of Statistical Natural Language Processing**. MIT Press, Cambridge, MA, USA, 1999.

[52] MATSUMOTO, T.; YIU, M. L. **Accelerating exact similarity search on cpu-gpu systems**. In: *2015 IEEE International Conference on Data Mining*, Nov 2015.

[53] MECHTAEV, S.; YI, J.; ROYCHOUDHURY, A. **Directfix: Looking for simple program repairs**. In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, p. 448–458, May 2015.

[54] MECHTAEV, S.; YI, J.; ROYCHOUDHURY, A. **Angelix: Scalable multiline program patch synthesis via symbolic analysis**. In: *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, p. 691–701, New York, NY, USA, 2016. ACM.

[55] MELUCCI, M. **Vector-Space Model**, p. 3259–3263. Springer US, Boston, MA, 2009.

[56] MIKOLOV, T.; CHEN, K.; CORRADO, G.; DEAN, J. **Efficient estimation of word representations in vector space**. *CoRR*, abs/1301.3781, 2013.

[57] MIKOLOV, T.; SUTSKEVER, I.; CHEN, K.; CORRADO, G.; DEAN, J. **Distributed representations of words and phrases and their compositionality**. *CoRR*, abs/1310.4546, 2013.

[58] NGUYEN, H. D. T.; QI, D.; ROYCHOUDHURY, A.; CHANDRA, S. **Semfix: Program repair via semantic analysis**. In: *2013 35th International Conference on Software Engineering (ICSE)*, p. 772–781, May 2013.

[59] PAN, J.; MANOCHA, D. **Fast gpu-based locality sensitive hashing for k-nearest neighbor computation**. In: *Proc. GIS*, 2011.

[60] PAULEVÉ, L.; JÉGOU, H.; AMSALEG, L. **Locality sensitive hashing: A comparison of hash function types and querying mechanisms**. *Pattern Recognition Letters*, 2010.

[61] PEI, Y.; FURIA, C. A.; NORDIO, M.; WEI, Y.; MEYER, B.; ZELLER, A. **Automated fixing of programs with contracts**. *IEEE Transactions on Software Engineering*, 40(5):427–449, May 2014.

[62] QI, Y.; MAO, X.; LEI, Y.; DAI, Z.; WANG, C. **The strength of random search on automated program repair**. In: *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, p. 254–265, New York, NY, USA, 2014. ACM.

[63] RAY, B.; HELLENDOORN, V.; GODHANE, S.; TU, Z.; BACCHELLI, A.; DEVANBU, P. **On the "naturalness" of buggy code**. In: *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, p. 428–439, New York, NY, USA, 2016. ACM.

[64] ROCHA, L.; RAMOS, G.; CHAVES, R.; SACHETTO, R.; MADEIRA, D.; VIEGAS, F.; ANDRADE, G.; DANIEL, S.; GONÇALVES, M.; FERREIRA, R. **G-knn: an efficient document classification algorithm for sparse datasets on gpus using knn**. In: *SIGAPP*. ACM, 2015.

[65] RONG, X. **word2vec parameter learning explained**. *CoRR*, abs/1411.2738, 2014.

[66] SAINI, I.; SINGH, D.; KHOSLA, A. **Qrs detection using k-nearest neighbor algorithm (knn) and evaluation on standard ecg databases**. *Journal of Advanced Research*, 4(4):331–344, 2013.

[67] SALTON, G.; WONG, A.; YANG, C.-S. **A vector space model for automatic indexing**. *Communications of the ACM*, 1975.

[68] SENGUPTA, S.; HARRIS, M.; GARLAND, M.; OWENS, J. D. **Efficient parallel scan algorithms for many-core gpus**. *Sci. Comp. with Multicore and Acc.*, p. 413–442, 2011.

[69] SHEN, Y.; TAN, S.; PAL, C. J.; COURVILLE, A. C. **Self-organized hierarchical softmax**. *CoRR*, abs/1707.08588, 2017.

[70] SHIM, K.; LEE, M.; CHOI, I.; BOO, Y.; SUNG, W. **Svd-softmax: Fast softmax approximation on large vocabulary neural networks**. In: Guyon, I.; Luxburg, U. V.; Bengio, S.; Wallach, H.; Fergus, R.; Vishwanathan, S.; Garnett, R., editors, *Advances in Neural Information Processing Systems 30*, p. 5463–5473. Curran Associates, Inc., 2017.

[71] SIENCNIK, S. K. **Adapting word2vec to named entity recognition**. In: *NODALIDA*, 2015.

[72] SIMONTON, T. M.; ALAGHBAND, G. **Efficient and accurate word2vec implementations in gpu and shared-memory multicore architectures**. In: *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, p. 1–7, Sept 2017.

[73] SISMANIS, N.; PITSIANIS, N.; SUN, X. **Parallel search of k-nearest neighbors with synchronous operations.** In: *HPEC*. IEEE, 2012.

[74] SKILLICORN, D. B.; TALIA, D. **Models and languages for parallel computation**. *ACM Comput. Surv.*, 30(2):123–169, June 1998.

[75] SOCHER, R.; BAUER, J.; MANNING, C. D.; NG, A. Y. **Parsing with compositional vector grammars**. In: *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, p. 455–465, Sofia, Bulgaria, Aug. 2013. Association for Computational Linguistics.

[76] TAN, S. H.; ROYCHOUDHURY, A. **relifix: Automated repair of software regressions**. In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, p. 471–482, May 2015.

[77] TAN, S. H.; YOSHIDA, H.; PRASAD, M. R.; ROYCHOUDHURY, A. **Anti-patterns in search-based program repair**. In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, p. 727–738, New York, NY, USA, 2016. ACM.

[78] TAYLOR, G.; BURMEISTER, R.; XU, Z.; SINGH, B.; PATEL, A.; GOLDSTEIN, T. **Training neural networks without gradients: A scalable admm approach**. may 2016.

[79] TRICENTIS. **Software fail watch: 5th edition**, 2017.

[80] WANG, Y.; SHRIVASTAVA, A.; WANG, J.; RYU, J. **Randomized algorithms accelerated over cpu-gpu for ultra-high dimensional similarity search**. SIGMOD '18. ACM, 2018.

[81] WEISS, C.; PREMRAJ, R.; ZIMMERMANN, T.; ZELLER, A. **How long will it take to fix this bug?** In: *Proceedings of the Fourth International Workshop on Mining Software Repositories*, MSR '07, p. 1–, Washington, DC, USA, 2007. IEEE Computer Society.

[82] XING, C.; WANG, D.; ZHANG, X.; LIU, C. **Document classification with distributions of word vectors**. In: *Signal and Information Processing Association Annual Summit and Conference (APSIPA), 2014 Asia-Pacific*, p. 1–5, Dec 2014.

[83] YEUNG, D.; DALLY, W. J.; AGARWAL, A. **How to choose the grain size of a parallel computer**.

[84] ZOJAJI, Z.; LADANI, B. T.; KHALILIAN, A. **Automated program repair using genetic programming and model checking**. *Applied Intelligence*, 45(4):1066–1088, 2016.