UNIVERSIDADE FEDERAL DE GOIÁS
INSTITUTO DE INFORMÁTICA

CHAYNER CORDEIRO BARROS

Acelerando a construção de tabelas hash para dados textuais com aplicações

Goiânia-GO
2020

UNIVERSIDADE FEDERAL DE GOIÁS
INSTITUTO DE INFORMÁTICA

TERMO DE CIÊNCIA E DE AUTORIZAÇÃO (TECA) 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 9.610/98, 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.

O conteúdo das Teses e Dissertações disponibilizado na BDTD/UFG é de responsabilidade exclusiva do autor. Ao encaminhar o produto final, o autor(a) e o(a) orientador(a) firmam o compromisso de que o trabalho não contém nenhuma violação de quaisquer direitos autorais ou outro direito de terceiros.

1. Identificação do material bibliográfico

[x] Dissertação [ ] Tese

2. Nome completo do autor

Chayner Cordeiro Barros

3. Título do trabalho

Acelerando a construção de tabelas hash para dados textuais com aplicações

4. Informações de acesso ao documento (este campo deve ser preenchido pelo orientador)

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

[1] Neste caso o documento será embargado por até um ano a partir da data de defesa. Após esse período, a possível disponibilização ocorrerá apenas mediante:
a) consulta ao(à) autor(a) e ao(à) orientador(a);
b) novo Termo de Ciência e de Autorização (TECA) assinado e inserido no arquivo da tese ou dissertação.
O documento não será disponibilizado 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.

Obs. Este termo deverá ser assinado no SEI pelo orientador e pelo autor.

Documento assinado eletronicamente por **Wellington Santos Martins**, **Professor do Magistério Superior**, em 17/12/2020, às 12:40, 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 **CHAYNER CORDEIRO BARROS**, **Discente**, em 17/12/2020, às 14:49, 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 **1759344** e o código CRC **17B33206**.

CHAYNER CORDEIRO BARROS

Acelerando a construção de tabelas hash para dados textuais com aplicações

Dissertação apresentada ao Programa de Pós–Graduação do Instituto de Informática da Universidade Federal de Goiás, como requisito parcial para obtenção do título de Mestre em Programa de Pós-Graduação em Ciência da Computação.

Área de concentração: Sistemas de Computação.

Orientador: Prof. Wellington Santos Martins

Goiânia-GO
2020

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

CORDEIRO BARROS, CHAYNER
Acelerando a construção de tabelas hash para dados textuais com aplicações [manuscrito] / CHAYNER CORDEIRO BARROS. - 2020.
XCIX, 99 f.

Orientador: Prof. Dr. WELLINGTON SANTOS MARTINS.
Dissertação (Mestrado) - Universidade Federal de Goiás, Instituto de Informática (INF), Programa de Pós-Graduação em Ciência da Computação, Goiânia, 2020.
Bibliografia.
Inclui tabelas, algoritmos, lista de figuras, lista de tabelas.

1. embeddings. 2. machine learning. 3. HPC. 4. matriz de coocorrência. 5. CUDA. I. SANTOS MARTINS, WELLINGTON, orient. II. Título.

CDU 004

UNIVERSIDADE FEDERAL DE GOIÁS

INSTITUTO DE INFORMÁTICA

ATA DE DEFESA DE DISSERTAÇÃO

Ata nº **25/2020** da sessão de Defesa de Dissertação de **Chayner Cordeiro Barros**, que confere o título de Mestre em Ciência da Computação, na área de concentração em Ciência da Computação.

Aos dezessete dias do mês de novembro de dois mil e vinte, a partir das catorze horas e trinta minutos, via sistema de webconferência da RNP, realizou-se a sessão pública de Defesa de Dissertação intitulada **"Acelerando a construção de tabelas hash para dados textuais com aplicações"**. Os trabalhos foram instalados pelo Orientador, Professor Doutor Wellington Santos Martins (INF/UFG) com a participação dos demais membros da Banca Examinadora: Professor Doutor Daniel Xavier de Sousa (IFG), membro titular externo; Professor Doutor Thierson Couto Rosa (INF/UFG), membro titular interno. A realização da banca ocorreu per meio de videoconferência, em atendimento à recomendação de suspensão das atividades presenciais na UFG emitida pelo Comitê UFG para o Gerenciamento da Crise COVID-19, bem como à recomendação de isolamento social da Organização Mundial de Saúde e do Ministério da Saúde para enfrentamento da emergência de saúde pública decorrente do novo coronavírus. Durante a arguição 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 Dissertação, 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 dezessete dias do mês de novembro de dois mil e vinte.

TÍTULO SUGERIDO PELA BANCA

Documento assinado eletronicamente por **Daniel Xavier de Sousa**, **Usuário Externo**, em 17/11/2020, às 16:54, 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 17/11/2020, às 16:55, 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 **Wellington Santos Martins**, **Professor do Magistério Superior**, em 17/11/2020, às 16:56, 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 **CHAYNER CORDEIRO BARROS**, **Discente**, em 17/11/2020, às 17:25, 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 **1658052** e o código CRC **D24C964B**.

Chayner Cordeiro Barros

Graduou-se em Engenharia da Computação na PUC-GO — Pontifícia Universidade Católica de Goiás. Durante sua graduação desenvolveu trabalhos de pesquisa na área de Computação Gráfica, Compiladores e Processamento de Imagens. Fez pós-graduação em Gestão de Projetos de Tecnologia da Informação. Atualmente trabalha com o desenvolvimento de soluções administrativas para os órgãos da Justiça Eleitoral.

Dedico esse trabalho em primeiro lugar a Deus, que iluminou meu caminho durante essa jornada. Dedico também à minha família, por acreditarem em mim e por sempre me darem o suporte necessário durante os períodos de dificuldade.

Agradecimentos

Sem a colaboração e o apoio de diversas pessoas, essa dissertação e o meu mestrado não teriam sido concluídos com sucesso.

Em primeiro lugar, quero agradecer o meu orientador, Professor Wellington Santos Martins, por toda paciência, empenho e direcionamento nos estudos e experimentos que realizei durante meu mestrado. Ao senhor, meu muitíssimo obrigado.

Quero agradecer os meus colegas do grupo de pesquisa, pelo apoio e amizade, durante esses anos de convívio.

Agradeço também aos demais professores, que me ensinaram e incentivaram tanto durante a realização das disciplinas que cursei, em especial à Prof.ª Nádia Félix Felipe da Silva, e aos funcionários do Instituto de Informática, em especial à Mariana Alves Rodrigues Santana, por sempre ser prestativa e atenciosa.

E, finalmente, agradeço à minha **família**, pelo amor e apoio incondicionais que sempre me deram durante todos os desafios que enfrentei na vida; o que de igual modo ocorreu durante a realização deste mestrado.

"Essentially, all models are wrong, but some are useful"

George E. P. Box,
Empirical Model-Building and Response Surfaces, p. 424, Wiley. ISBN 0471810339.

Resumo

Barros, Chayner Cordeiro. **Acelerando a construção de tabelas hash para dados textuais com aplicações**. Goiânia-GO, 2020. 98p. Dissertação de Mestrado. Instituto de Informática, Universidade Federal de Goiás.

A mineração de texto (*text mining*) se caracteriza pela extração de informações a partir de dados textuais, nos mais diversos formatos, objetivando a produção de conhecimento, a classificação, clusterização, tradução desta informação entre outras tarefas. Para que a mineração de textos seja eficiente alguns procedimentos são realizados sobre os dados para garantir que eles contenham apenas conteúdo relevante à análise que será realizada, e que estejam estruturados num formato mais fácil de ser manipulado computacionalmente. Diversas tarefas de pré-processamento devem ser realizadas sobre esses dados, para alcançar a qualidade e a representação desejada. Neste sentido, o presente trabalho propõe uma implementação de tabela *hash* capaz de explorar o alto paralelismo disponível nas GPUs de forma eficiente, como forma de aumentar o desempenho das tarefas de pré-processamento de texto. Entretanto, este trabalho não apenas apresenta algoritmos mais eficientes, mas também demonstra a viabilidade de seu uso em aplicações como a geração da matriz de coocorrência e da representação do texto utilizando-se *embeddings*.

Palavras–chave

computação de alto desempenho, aprendizado de máquina, mineração de texto, matriz de coocorrência, tabelas *hash*, *embeddings*, CUDA

Abstract

Barros, Chayner Cordeiro. **Accelerating the construction of hash tables for textual data with applications**. Goiânia-GO, 2020. 98p. MSc. Dissertation. Instituto de Informática, Universidade Federal de Goiás.

Text mining is characterized by the extraction of information from textual data, in the most diverse formats, aiming at the knowledge production, classification, clusterization, translation of this information among other things. In order for text mining to be efficient, some procedures are performed on the data to ensure that it contains only content relevant to the analysis to be performed, and that it is structured in a format that is easier to manipulate computationally. Several pre-processing tasks must be performed on this data, in order to achieve the desired quality and representation. In this sense, the present work proposes an implementation of a hash table capable of efficiently exploring the high parallelism available in GPUs, as a way to increase the performance of pre-processing tasks. However, this work not only presents more efficient algorithms, but also demonstrates the feasibility of its use in applications such as the generation of the co-occurrence matrix and the representation of the text using embeddings.

Keywords

high performance computing, hpc, machine learning, text mining, co-occurrence matrix, hash tables, embeddings, CUDA

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Algoritmos

Lista de Códigos de Programas

CAPÍTULO 1

Introdução

O advento da Internet permitiu, não apenas que pessoas ao redor do mundo se conectassem, mas também possibilitou que estas mesmas pessoas pudessem se expressar de formas antes inconcebíveis. Redes sociais, salas de bate-papo, comentários em notícias e postagens assim que essas são publicadas, livros, revistas, artigos científicos etc; são alguns exemplos de como podemos compartilhar o pensamento individual e coletivo nos dias de hoje. Com tantos dados disponíveis, gerados pelas mais distintas fontes, uma área de pesquisa tem sido alvo de interesse dos pesquisadores e, consequentemente, das empresas que manipulam este volume de dados: a extração de conhecimento.

A extração de conhecimento, também conhecida como KDD (*Knowledge Discovery in Databases*) é um processo onde o conhecimento é obtido a partir de grandes volumes de dados. Para obter tal conhecimento, uma sequência de fases foi estabelecida, iniciando-se com a coleta dos dados, seguida do tratamento destes dados e, por fim, a construção de uma apresentação do resultado da extração realizada. O processo de KDD é dividido em cinco fases: seleção de dados, pré-processamento, transformação, mineração e interpretação/avaliação. É um processo iterativo, onde as fases podem ser repetidas continuamente em busca de melhores resultados, e também pode ser considerado um processo interativo, onde os usuários participam da extração. Embora cada fase do processo KDD seja independente, podendo ser realizadas individualmente, há uma relação de dependências entre elas. E apesar do nome, a origem dos dados não necessariamente é uma base de dados estruturada [41] [19].

Este conhecimento é um ativo valioso para empresas, porque pode garantir a liderança dentro do seu segmento de atuação. Por exemplo, ao investigar os comentários emitidos por seus clientes, uma loja de varejo online pode determinar quais produtos são mais atraentes e quais não são do agrado de seus clientes [52]. Desta forma, é possível otimizar o catálogo de produtos oferecidos, e até mesmo determinar o motivo pelo qual os clientes estão insatisfeitos com a loja ou seus produtos (entregas que sempre atrasam, por exemplo).

Para que a extração de conhecimento seja eficiente alguns procedimentos são realizados sobre os dados para garantir que eles contenham apenas conteúdo relevante

à análise que será realizada, e que estejam estruturados num formato mais fácil de ser manipulado computacionalmente. Estes procedimentos são denominados de "pré-processamento" e constituem uma fase crucial no processo de extração.

Durante o pré-processamento são removidos possíveis ruídos, os dados são normalizados e agrupados de acordo com sua relevância para a análise. No caso da extração de conhecimento, os ruídos são, em sua maioria, representados por termos irrelevantes, que pouco ou nada acrescentam à ideia que é transmitida no texto; a normalização é feita pela remoção de afixos nos termos encontrados no documento; e a relevância pode ser determinada com o uso de técnicas que analizam desde a frequência da ocorrência do termo até as relações semânticas entre eles. Para auxiliar nessa tarefa, é importante escolher um formato adequado para representar os termos de forma estruturada, que seja facilmente manipulável computacionalmente. Essa representação é o objetivo deste trabalho.

Existem diversos modelos de representação de texto, que foram propostos nas últimas décadas. Estes modelos divergem tanto no formato interno de representação da informação (vetores densos ou esparsos, por exemplo) quanto nos elementos que eles representam: sub-unidades dos termos (letras, sílabas ou outro agrupamento), termos inteiros, agrupamentos de termos (n-grams), parágrafos, documentos inteiros. Cada um destes modelos propõe resolver o problema de representação da informação de uma forma diferente.

1.1 Objetivos

O objetivo principal desta pesquisa é a produção de um algoritmo paralelo eficiente para a construção de tabelas *hash*, que se beneficie das principais características da GPU para alcançar um alto desempenho, mas que seja aplicável também em implementações para a CPU. Essa estrutura de dados permitirá que estratégias de processamento de texto existentes se beneficiem do ganho de desempenho obtido na sua construção. Viabilizando, assim, a reconstrução "online" dos elementos de processamento de texto que fizerem uso desta estrutura, como, por exemplo, os *embeddings*, que beneficiariam-se desta capacidade, mantendo vivas as relações entre os termos existentes, permitindo a expansão do espaço vetorial com a adição de novos termos dinamicamente.

Para que o objetivo seja alcançado, foram definidas algumas metas específicas:

- Identificar os modelos de representação existentes, para que seja possível definir parâmetros comparativos;
- Propor algoritmos paralelos para acelerar o pré-processamento de bases textuais;
- Determinar uma abordagem eficiente de manipulação e representação de documentos na memória de uma GPU;

- Implementar essa abordagem e efetuar comparações com os algoritmos identificados anteriormente, comparando a acurácia e o tempo de execução obtidos;
- Comparar os resultados obtidos com a implementação da abordagem em GPU proposta com os resultados obtidos com o algoritmo original, apresentando a acurácia e os tempos de execução obtidos;

1.2 Justificativas

Ao analisar alguns trabalhos relacionados ao processamento de linguagem natural e à extração de conhecimento, percebeu-se que pouca ou nenhuma ênfase é dada ao pré-processamento das bases de texto necessárias para essas tarefas. Com exceção de trabalhos voltados especificamente ao pré-processamento, a forma e os métodos utilizados não são descritos, e durante o levantamento bibliográfico foram encontrados poucos trabalhos relatando formas eficientes de se realizar o pré-processamento.

Associe-se a isto o fato de que os trabalhos acadêmicos comumente são realizados sobre bases de dados "prontas", previamente trabalhadas e em formatos específicos. O uso de tais bases é importante porque permite, principalmente, que outros pesquisadores consigam repetir os passos realizados pelos autores dos trabalhos, confirmando, assim, a efetividade de suas propostas. Porém, a menor disponibilidade de bases diversificadas impõe limitações à realização de pesquisas em cenários diferentes. É o caso de bases na língua portuguesa que podem ser consideradas escassas, se comparadas com a quantidade de bases disponíveis na língua inglesa. Se considerarmos bases especializadas, como *corpora* na área médica, por exemplo, essa escassez se torna ainda mais evidente. Essa limitação é imposta, consequentemente, aos produtos e subprodutos destas bases de dados textuais, como, por exemplo, os *embeddings*.

Ao analisar a situação exposta, vislumbrou-se algumas alternativas para auxiliar pesquisadores e especialistas na construção de soluções que façam uso de bases textuais, seja para teste, treinamento e até mesmo em produção. As alternativas propostas formam um conjunto de algoritmos capazes de realizar as atividades de pré-processamento e construção de *embeddings* em paralelo, utilizando a arquitetura de alta performance disponível nas GPUs.

1.3 Contribuições

Enumera-se como as principais contribuições deste trabalho:

1. A proposição de uma abordagem multiplataforma para a implementação de tabela *hash*, uma estrutura de dados dinâmica e de baixa complexidade computacional,

tanto no tempo quanto no espaço;

2. A demonstração da viabilidade do uso de arquiteturas *manycore* para a realização de tarefas de pré-processamento, como a limpeza do texto, a geração do vocabulário e das estatísticas de coocorrência, que foi realizada em trabalho publicado anteriormente [7];
3. A demonstração da viabilidade do uso de tabelas *hash* e matriz de coocorrência para a construção acelerada por GPU de representações vetoriais dos elementos do texto;
4. A apresentação de um estudo comparativo entre os principais algoritmos utilizados para a construção de *embeddings* e matriz de coocorrência.

1.4 Organização do documento

Os capítulos seguintes a este encontram-se organizados da seguinte forma: no Capítulo 2 são apresentados os conceitos relevantes para a pesquisa sobre representações de bases textuais e aprendizado de máquina, como também são apresentados os principais trabalhos relacionados à proposta deste trabalho; no Capítulo 3 são descritos os algoritmos propostos neste trabalho; no Capítulo 4 são demonstrados os experimentos e análises realizados e os resultados alcançados; e, por fim, no Capítulo 5 são apresentadas as conclusões sobre os resultados obtidos e uma proposta para o prosseguimento deste trabalho.

CAPÍTULO 2

Embasamento Teórico e Trabalhos Relacionados

2.1 Pré-Processamento

Conforme dito anteriormente, o pré-processamento é uma etapa vital na mineração de textos, pois remove redundâncias, e transforma os termos de um *corpus* numa representação computacional mais eficiente.

Para apresentar os passos necessários, tome-se como exemplo a aplicação de filtragem de *spam* dos e-mails. A caixa de mensagens do usuário representa o que é denominado *corpus*, contendo todo o conjunto de documentos (mensagens de e-mail) que deverão ser analisados e classificados. Logo, o objetivo da filtragem de *spam* nada mais é que classificar as mensagens entre duas categorias distintas: *spam* e não *spam*. Desta forma, processa-se cada mensagem individualmente, extraindo as palavras (ou termos) existentes dentro da mensagem[1]. Esta extração é denominada *tokenization* (*tokenização*), e seu objetivo é extrair os termos que compõem o texto. Os termos podem ser extraídos e colocados numa lista, por exemplo.

Após a tokenização, os termos são filtrados, removendo-se aqueles que podem ser considerados irrelevantes. Tais termos são comumente denominados *stopwords*, e o conjunto que os define é dependente do idioma em que o texto foi escrito e do propósito da aplicação que os está analisando. É geralmente um passo simples de ser executado, mas que muitas vezes pode ser dispendioso, dependendo da forma como é implementado, visto que para cada termo encontrado durante a tokenização, uma varredura completa deverá ser realizada no conjunto de termos irrelevantes (*stopwords*).

Assim que os *stopwords* são removidos do conjunto global de termos, efetua-se sua normalização, retirando os afixos com o intuito de encontrar os radicais das palavras. Este passo é conhecido como *stemming*, e é um passo importante porque o sentido

[1] Um filtro de *spam* pode analisar outras informações além do texto da mensagem, como cabeçalhos, endereços IP, etc. Para simplificar a exemplificação, utilizou-se apenas o texto da mensagem.

da palavra é geralmente passado no radical, podendo as variações apresentadas por suas derivações serem consideradas irrelevantes para uma análise computacional, dependendo da aplicação. Para exemplificar este conceito, observe os termos "conhecimento", "conhecer" e "conhecia". É possível perceber que ao se encontrar qualquer um destes termos num texto, a noção que representam é a de "conhecimento". Portanto, ao invés de analisarmos individualmente todos os termos que podem representar o mesmo conceito, simplesmente os sintetizamos num termo menor, que carrega a noção desejada e que representa todos estes. Existem diversos algoritmos para extração de radicais, muitos deles dependentes do idioma do texto analisado, como é o caso do Porter [56], bastante utilizado para textos em inglês. Alguns possuem implementações para os idiomas mais comuns, inclusive o português, como é o caso do algoritmo construído em *Snowball*[2] [57]. Existem também algoritmos capazes de generalizar o conceito de radicalização ao ponto de obterem os radicais independentemente do idioma analisado [51]. No caso dos termos usados como exemplo, o radical obtido pelo algoritmo em *Snowball* é "conhec" [64] [35] [66].

Assim como a remoção de *stopwords*, o *stemming* é um passo opcional, sendo executado ou não, de acordo com o objetivo da aplicação. Existe uma variante para o *stemming*, denominada *lemmatization*. Ambas as técnicas são utilizadas para reduzir para uma base comum formas flexionadas, e algumas vezes, formas derivadas de palavras. Enquanto a radicalização ocupa-se primariamente com a remoção de sufixos através de um processo heurístico, a lematização refere-se a uma análise mais elaborada das palavras, considerando características morfológicas destas, removendo flexões e encontrando sua forma mais básica, também conhecida como *lema*, que é a forma como a palavra seria encontrada em dicionários [54] [30]. Em [6] as autoras apontam que a lematização produz melhores resultados se comparada com a radicalização. A comparação é realizada sobre termos na língua inglesa. De acordo com as próprias autoras o aumento na precisão é insignificante.

Após a radicalização (*stemming*) dos termos, o pré-processamento ocupar-se-á da representação computacional deste termo. O termo pode ser representado de diversas formas, e o formato é totalmente dependente da aplicação. Este formato pode ser exatamente o mesmo utilizado até o momento nos passos de pré-processamento exemplificados anteriormente, isto é, uma *string* de caracteres representando as letras que compõem o termo. Contudo, esta representação não é considerada ideal por diversos motivos, incluindo o fato de que cada caractere normalmente possui um código diferente dependendo da codificação utilizada para representar o texto. Ou seja, o código ASCII

[2] *Snowball* é uma linguagem de programação criada por Martin Porter, autor do famoso algoritmo de radicalização *Porter*. Nela é possível descrever um algoritmo de *stemming* num script, que será posteriormente transcrito para um código ANSI C ou Java. Site oficial: http://snowballstem.org/

que representa a letra “a” é diferente do código EBCDIC[3] que representa a mesma letra.

2.2 Modelos de Representação Textual

Para representar dados num computador devemos escolher dentre os vários modelos possíveis para o formato e tipo de dados que se quer representar. Por exemplo, para representar números inteiros o modelo de representação adotado é uma sequência de bits agrupados e ordenados por relevância, com tamanho múltiplo de 2. Desta forma, o número 42 é representado como a sequência de bits 00101010, agrupados em 8 bits. Existem diversos modelos usados para representar diferentes tipos de dados, como números em ponto flutuante, strings de caracteres, imagens, sons etc. Todos estes modelos são derivados de representações mais simples, como o mostrado para o número 42. Essa derivação é importante porque nativamente o computador compreende e trabalha internamente com números binários (os 0s e os 1s). Portanto, encontrar um modelo computacional que represente os dados desejados num formato mais acessível para a unidade de processamento (seja uma CPU ou uma GPU) é imprescindível para que a análise sobre os dados seja a mais eficiente computacionalmente possível.

Conforme dito anteriormente, um termo é uma sequência de caracteres, representada normalmente em memória por um *array* de números de comprimento fixo (8 bits, por exemplo), onde cada valor numérico está associado com um caractere específico. Um exemplo de tal representação (também denominada de “codificação” ou “*encoding*”) é a tabela ASCII, amplamente utilizada em computadores pelo mundo todo, e que consegue representar vários caracteres de controle, como ENTER, SPACE, etc., os dígitos de 0 a 9 e as letras que compõem o nosso alfabeto. Devido ao grande número de codificações e idiomas existentes, é inviável utilizar a representação dos termos como *strings* de caracteres (Figura 2.1). Assim, normalmente os termos são representados por um número inteiro que os distingue entre os demais termos que compõem o vocabulário da coleção de documentos sendo analisada. Ou seja, para um vocabulário contendo N termos, cada termo será representado por um inteiro entre $0 \leq i \leq N$. Fatores inerentes a aplicação também devem ser considerados, como é o caso de algoritmos de aprendizado de máquina e NLP, por exemplo, que trabalham melhor se suas variáveis de entrada e saída tiverem um tamanho fixo e bem definido [22].

Para melhor estruturar um documento a ser processado computacionalmente, costuma-se associar um peso (valor) a cada termo, indicando sua importância no documento. O BOW (*Bag of Words* — ou sacola de palavras) é um dos modelos mais simples

[3] EBCDIC é uma codificação de caracteres com suporte completo ao conjunto de caracteres derivados do latim. É usado especialmente em *mainframes* da IBM.

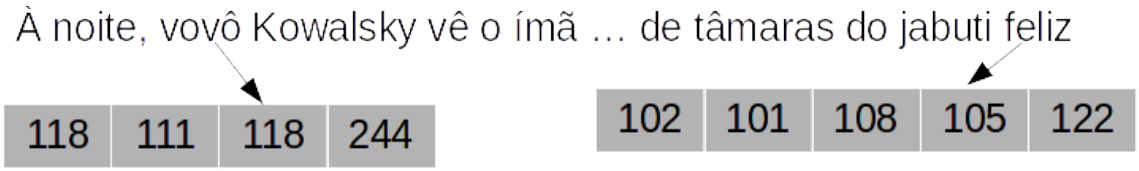


Figura 2.1: *Exemplo de palavras na codificação ASCII*

para esta tarefa [36][70]. Ele geralmente associa cada termo à frequência com que aparece no documento (*term-frequency – TF*). A quantidade de vezes que cada termo/palavra aparece no documento define, de modo geral, sua relevância no documento. Existem outras características que identificam a relevância de um termo com maior acurácia do que apenas a sua frequência. O TF-IDF (*term frequency/inverse document frequency*), por exemplo, é uma medida que contabiliza a frequência do termo sobre o inverso de sua frequência dentro de todos os documentos do *corpus*. A ideia por trás dessa abordagem é que as palavras que ocorrem com maior frequência não agregam "informação" suficiente ao modelo [37]. Em tarefas onde palavras são consideradas *features*, o modelo BOW pode ser utilizado para criar um vetor destas *features*, quando este número não é inicialmente conhecido e quando a ordem destas palavras não é relevante. Uma forma de representar essas palavras é utilizar um vetor esparso, denominado *one-hot*, cuja dimensão é igual ao tamanho do vocabulário, com uma entrada contendo o valor 1, representando a palavra, e zeros em todas as demais posições.

Para exemplificar, suponha que o *corpus* seja formado pela frase "À noite, vovô Kowalsky vê o ímã cair no pé do pinguim queixoso e vovó põe açúcar no chá de tâmaras do jabuti feliz."[4]. A representação *one-hot encoding* é formada por vetores com dimensões equivalentes à quantidade de termos únicos do vocabulário, sendo, neste caso 26 *tokens*, conforme Figura 2.2.

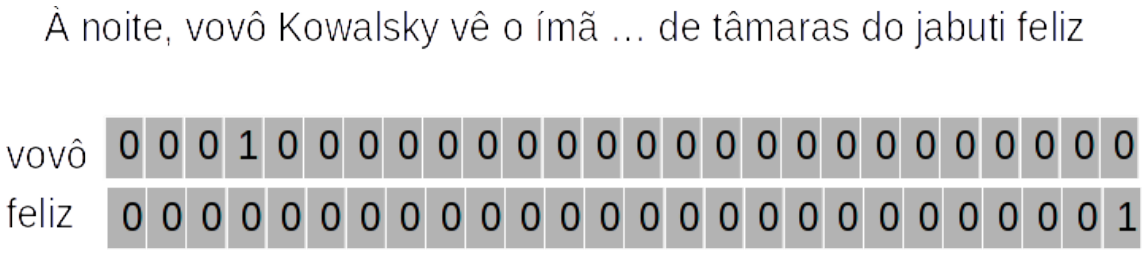


Figura 2.2: *Exemplo de vetores one-hot*

Retomando o exemplo do filtro de *spam* apresentado anteriormente, armazena-se todas as palavras contidas dentro da mensagem de e-mail dentro de uma *bag-of-words*. Se este filtro de *spam* for implementado como um classificador Bayesiano[59][4], então este poderia ser modelado como uma coleção de palavras construída a partir de uma entre

[4]Exemplo de pangrama da lingua Portuguesa, com 90 letras, incluindo todas as letras acentuadas.

duas distribuições probabilísticas possíveis. A primeira representada por uma *bag-of-words* contendo as palavras mais comuns em mensagens consideradas *spam*, e a segunda, uma *bag-of-words* com palavras encontradas em mensagens legítimas. Para determinar se uma determinada mensagem é *spam* o classificador basicamente analisa a probabilidade da mensagem pertencer a *bag-of-words* de palavras relacionadas a *spam*. Por exemplo, se uma mensagem qualquer possuir 57% de termos da *bag-of-words* de *spam*, então a chance desta mensagem ser um *spam* é maior do que uma mensagem legítima. Observe que este limiar não é fixo ou válido para todos os casos.

Outra abordagem usada para capturar informações contextuais dos termos é a utilização de *n-grams*[36][10]. Nessa abordagem associa-se n termos, de modo que estes são analisados em conjunto. Os *n-grams* podem ser formados por fonemas, sílabas, letras, palavras ou quaisquer outros elementos textuais necessários para uma dada aplicação. Diferentemente da abordagem usada tradicionalmente com *bag of words*, onde cada termo é analisado isoladamente, no *n-gram* leva-se em consideração a vizinhança do termo, sendo, portanto, possível armazenar informação espacial do termo dentro de um documento. Conceitualmente, o modelo *bag of words* pode ser considerado um caso especial do modelo *n-gram*, com $n = 1$ e os termos representando palavras inteiras.

Um modelo que vem sendo bastante utilizado baseia-se na arquitetura *skip-gram*. *Skip-grams* são uma generalização do modelo *n-gram* apresentado anteriormente, nos quais os componentes que compõem o modelo não são necessariamente consecutivos dentro do texto, podendo haver intervalos vazios (*gaps*) que podem ser saltados (*skipped over*). Esta abordagem permite superar uma dificuldade encontrada com a esparsidade de dados normalmente encontrada na análise de *n-grams* tradicional. Alguns dos modelos computacionais que se baseiam no *skip-gram* são o *word2vec*[45], o *GloVe*[53] e o *fasttext*[10]. O *word2vec* e o *GloVe* são similares, sendo uma das diferenças notáveis o fato de que o *word2vec* é um modelo preditivo, enquanto o *GloVe* é um modelo baseado em contagem. O *fasttext* é uma variação do *word2vec* utilizando *n-grams*.

2.2.1 Modelos de Word Embeddings

Word Embeddings são modelos que representam informações num espaço vetorial, usando uma semântica distributiva. Estes modelos vêm sendo utilizados desde o ano de 1990 [16]. Mas o termo foi cunhado por Bengio at al.[8], que treinou esses *embeddings* e os parâmetros do modelo num modelo neural de linguagem.

Collobert e Weston [13] conseguiram demonstrar as vantagens de se utilizar *word embeddings* pré-treinados. Eles não apenas estabeleceram os *word embeddings* como uma ferramenta útil, mas também introduziram uma arquitetura de rede neural que formou a base para muitas das abordagens utilizadas atualmente.

Contudo, pode-se dizer que a popularização dos *word embeddings* veio com a publicação de dois trabalhos de Mikolov ([45], [46]) e com a criação do *word2vec*.

Os modelos de *word embeddings* foram derivados dos modelos de linguagens (*language models*) usados em NLP, que possuem a habilidade de aprender uma distribuição probabilística das palavras de um vocabulário V. Para auxiliar na descrição dos modelos, assume-se que existe um *corpus* de treinamento que contém uma sequência T de palavras $w_1, w_2, w_3, \ldots, w_T$ que pertecem a um vocabulário V cujo tamanho é $|V|$. Considera-se como contexto um grupo de n palavras.

Esses modelos usados em NLP computam a probabilidade de uma palavra w_t ocorrer dada a ocorrência das $n-1$ palavras anteriores. Ou seja, $P(w_t|w_{t-1}, \ldots, w_{t-n+1})$. Logo, podemos aproximar o resultado de uma sentença ou documento através do produto das probabilidades de cada palavra ocorrer, dada a ocorrência de n palavras anteriores:

$$P(w_1, \ldots, w_T) = \prod_i P(w_i|w_{i-1}, \ldots, w_{i-n+1})$$

Essa aproximação pode ser adaptada para modelos que utilizam n-grams[5]. Para isso calcula-se a probabilidade de uma palavra baseada nas frequências dos n-grams que a constituem:

$$P(w_t|w_{t-1}, \ldots, w_{t-n+1}) = \frac{count(w_{t-n+1}, \ldots, w_{t-1}, w_t)}{count(w_{t-n+1}, \ldots, w_{t-1})}$$

Utilizando-se redes neurais, podemos obter o mesmo resultado através do uso de uma camada conhecida como *softmax*[6]:

$$P(w_t|w_{t-1}, \ldots, w_{t-n+1}) = \frac{\exp(h^\top v\prime_{w_t})}{\sum_{w_i \in V} \exp(h^\top v\prime_{w_i})}$$

O produto $h^\top v\prime_{w_t}$ representa o cálculo da probabilidade da palavra w_t, que será normalizada pela soma das probabilidades de todas as palavras em V. E h representa o vetor resultante da camada oculta na rede neural na figura 2.3. Como pode ser visto nessa figura, a saída desta camada ($v\prime_w$) é o *embedding* da palavra w, que corresponde

[5] n-gram é um agrupamento de n itens formado a partir de uma palavra ou texto. n-grams permitem uma granularidade mais fina na representação do texto, visto que podemos delimitar o tamanho máximo da unidade representada. Desta forma, podemos representar sílabas, fonemas, letras ou palavras inteiras. Essa representação difere de outras como o BOW, onde apenas palavras inteiras são consideradas.

[6] A função *softmax* é uma generalização da função logística que comprime um vetor de K-dimensões num vetor de K-dimensões com $\sigma(z)$ valores reais, onde cada entrada deste vetor está dentro do intervalo $(0,1]$ e a soma de todos os valores não pode ultrapassar 1. O resultado da função *softmax* pode ser usado para representar uma distribuição probabilística de K diferentes possíveis resultados. Ela costuma ser utilizada para classificar um valor quando este valor pode estar em mais de uma classe. Exemplos de métodos que utilizam *softmax* incluem *multinomial logistic regression*, *naïve Bayes*, e redes neurais artificiais.

à matriz de pesos da camada *softmax* da rede. É possível observar que apesar do fato de $v\prime_w$ representar a palavra w, ele não é aprendido conjuntamente com o *embedding* v_w da palavra de entrada.

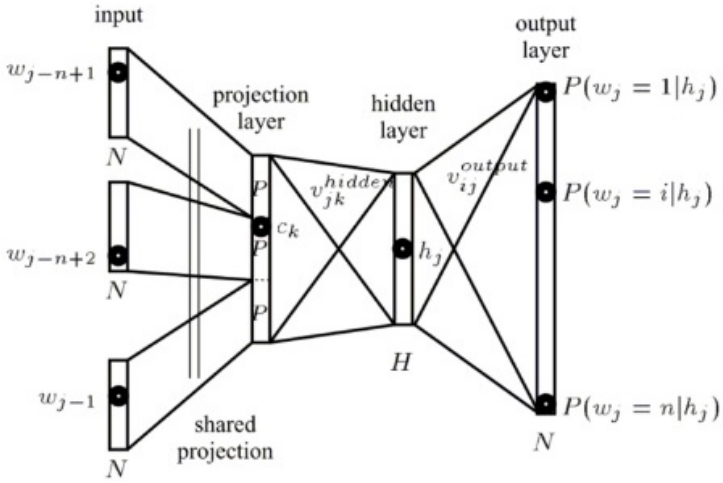


Figura 2.3: *Modelo de linguagem neural [9]*

Analisando novamente a figura podemos observar que iremos calcular na camada de saída da rede neural a probabilidade de cada palavra w. E nosso objetivo é realizar essa operação de forma eficiente. Para isso, calculamos a multiplicação da matriz h e da matriz de pesos, cujas linhas são formadas pelos $v\prime_w$ de todas as palavras w em V. Então, usamos o vetor resultante desta operação (também conhecido como *logit*) como entrada para a função *softmax*, que por sua vez "comprime" esse vetor numa distribuição probabilística em relação a todas as palavras de V.

É possível observar que uma rede neural *feed forward* que analisa palavras w de um vocabulário V como entrada e as trata como vetores dentro de um espaço de menor dimensão, e repetidamente melhora a aproximação dessa representação através de *back-propagation*, está produzindo *word embeddings* como pesos para a primeira camada, que é algumas vezes é denominada *embedding layer*.

A principal diferença entre uma rede neural que produz *word embeddings* como um subproduto e um método, como o *word2vec*, cujo objetivo explícito é a geração de *word embeddings* está na complexidade computacional. Gerar *word embeddings* com uma arquitetura muito profunda (muitas camadas) é simplesmente muito dispendioso, computacionalmente falando, quando usamos um vocabulário muito grande. Esta é a principal razão porque se levou tanto tempo ([45]) para que os *word embeddings* fossem amplamente utilizados em NLP. A complexidade computacional é um fator-chave para os modelos de *word embeddings*.

Outra diferença está no resultado esperado do treinamento. *word2vec* é movido pela produção de *word embeddings*, que codificam relações semânticas genéricas, que são importantes para muitas aplicações. Desta forma, *word embeddings* treinados desta

maneira não serão úteis para aplicações que não se baseiam neste tipo de relacionamento. Por outro lado, redes neurais geralmente produzem *embeddings* específicos da tarefa que estão tratando e que são de uso limitado em outras atividades. Observe que uma tarefa que depende de representações semantaticamente coerentes, tais como modelagem de linguagens, produzirão *embeddings* semelhantes aos modelos *word embeddings*. De um modo geral, *word2vec* e outras representações vetoriais variantes, como o *GloVe* podem ser usadas como uma espécie de "inicialização dos pesos" que produzem características úteis sem a necessidade de treinamentos muito longos.

Matriz de Coocorrência

De um modo geral existem duas classificações de *embeddings* utilizadas atualmente: *frequency-based* e *prediction-based*.

Os modelos baseados em frequência (*frequency-based*) correspondem àqueles que são construídos exatamente como o nome diz, pela contagem da frequência dos termos existentes e a correlação entre eles. Enquanto que os modelos baseados em predição (*prediction-based*) são aqueles treinados, normalmente utilizando uma rede neural para esta tarefa, onde a correlação entre os termos é determinada a partir de uma função de probabilidade.

Esta seção descreve as características de uma matriz de coocorrência, peça fundamental para a construção de *embeddings* baseados em frequência. Em seguida são apresentados os principais modelos preditivos.

A ideia por trás da matriz de coocorrência é a de que termos semelhantes tendem a aparecer próximos dentro do texto e possuírem um contexto semântico similar. A coocorrência diz respeito a um par de palavras que aparecem juntas diversas vezes dentro do que é denominado "janela de contexto".

Observe na Figura 2.4 um trecho de documento com o seguinte texto: "*À noite, vovô Kowalsky vê o ímã cair no pé do pinguim...*". Para uma janela simétrica de tamanho 2, temos dois termos à direita e dois à esquerda que devem ser analisados. Para cada *par* encontrado, um contador para esse par, na matriz de co-ocorrência deve ser incrementado. Os pares são representados na figura, e os termos centrais são representados por retângulos com fundo azul claro. Por exemplo, para a palavra "À", a janela de contexto associa os termos "noite" e ",". No exemplo apresentado, o sinal de pontuação está sendo considerado, mas pode ser ignorado dependendo da aplicação.

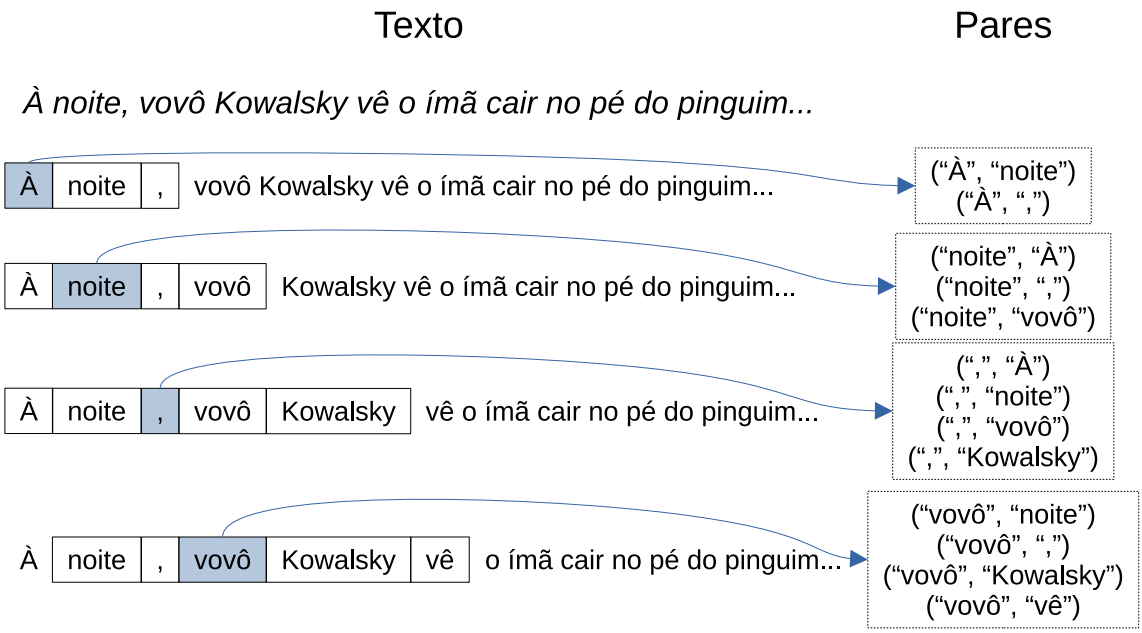
A matriz de coocorrência possui diversas vantagens dentre as quais pode-se citar:

- Preserva as relações semânticas entre as palavras, visto que palavras como "homem" e "mulher" tendem a estar próximas;

- Utiliza fatoração, que é um problema bem definido e que pode ser resolvido eficientemente;
- Pode ser reaproveitada, isto é, uma vez computada pode ser reutilizada e atualizada, podendo ser considerada extremamente rápida.

Contudo, a matriz de coocorrência exige grandes quantidades de memória, visto que grande parte de seu conteúdo será preenchido por zeros, porque a maioria das palavras de uma língua jamais aparecerá no mesmo contexto.

Figura 2.4: *Exemplo de janela de contexto simétrica de tamanho 2.*



Fonte: Chayner Cordeiro Barros

GloVe

Pennington et al. [53] também desenvolveram um algoritmo para a construção de *embeddings* pré-treinados. Assim como no Word2Vec, o principal objetivo é construir uma representação vetorial capaz de capturar a semântica do texto usado no treinamento. Enquanto o Word2Vec utiliza uma rede neural para obter os vetores como um subproduto da rede, o GloVe emprega uma análise estatística sobre os contadores de frequência dos termos do corpus. Apesar dessa diferença, os resultados obtidos por ambos os algoritmos são praticamente os mesmos.

A ideia de trabalhar com contadores globais já era usada muito antes do Word2Vec ou o GloVe serem desenvolvidos, sendo que algoritmos como o LSA (*Latent Semantic Analysis*) fazem uso dessas matrizes para converter palavras em vetores.

Contudo, os vetores obtidos pelo LSA, por exemplo, não possuem as mesmas características que os vetores produzidos pelo GloVe ou o Word2Vec. Os vetores desses últimos podem ser usados para expressar analogias entre termos através de operações aritméticas entre os vetores ($king - man + woman = queen$).

Ao utilizar uma matriz de co-ocorrência, o GloVe tem que tratar o problema causado pelo fato de que as palavras mais frequentes contribuem de forma desproporcional para o cálculo da similaridade, e por isso é necessário realizar uma normalização. Para isso, ao invés de utilizar os valores absolutos dos contadores de frequência presentes nessa matriz, o algoritmo utiliza uma distribuição probabilística calculada com o auxílio da matriz PPMI (*Positive Pointwise Mutual Information*).

O PPMI é uma adaptação do PMI (*Pointwise Mutual Information*). O PMI, quando aplicado neste cenário, representa a probabilidade de um termo co-ocorrer com outro, considerando as probabilidades de ocorrência dos termos individualmente:

$$pmi(x;y) = \log \frac{p(x,y)}{p(x) \cdot p(y)} = \log \frac{p(x|y)}{p(x)} = \log \frac{p(y|x)}{p(y)} \tag{2-1}$$

O PPMI ignora os logaritmos negativos, que representam termos com pouca relevância no documento:

$$ppmi(x;y) = \max pmi(x,y), 0 \tag{2-2}$$

Dessa forma, a relação entre as palavras pode ser obtida analisando-se a razão das probabilidades de co-ocorrência entre elas e um grupo de palavras de teste. No trabalho original, esse grupo é denominado k. Então, seja $P(k|w)$ a probabilidade de que a palavra k apareça no contexto da palavra w. Seguindo o exemplo oferecido no artigo[53], considere a palavra *solid* que tem forte relação com a palavra *ice* no exemplo demonstrado. Visto que há uma relação semântica direta entre as duas palavras, $P(solid|ice)$ terá, consequentemente, um valor alto, enquanto que $P(solid|steam)$ teria um valor muito baixo. Assim, a razão entre $P(solid|ice)/P(solid|steam)$ também será um valor alto. Se considerarmos agora a palavra *gas* que é relacionada a *steam*, mas não tem relação direta com *ice*, a razão entre $P(gas|ice)/P(gas|steam)$ seria um valor extremamente pequeno.

Para descartar ruídos, isto é, palavras que raramente coocorrem e que não possuem uma carga semântica representativa, os pesquisadores empregaram a equação 2-3 para definir pesos às relações encontradas:

$$f(x) = \begin{cases} (x/x_{max})^{\alpha}, & \text{se } x < x_{max} \\ 1, & \text{caso contrário} \end{cases} \tag{2-3}$$

A Figura 2.5 apresenta o gráfico demonstrando o ponto de corte obtido, ao aplicar a Equação 2-3 com o peso $\alpha = \frac{3}{4}$. Desta forma, palavras acima desse limiar serão

automaticamente ignoradas, devido à baixa relação de coocorrência entre elas.

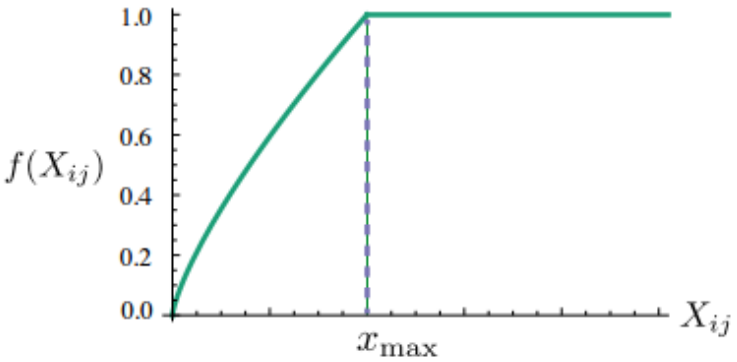
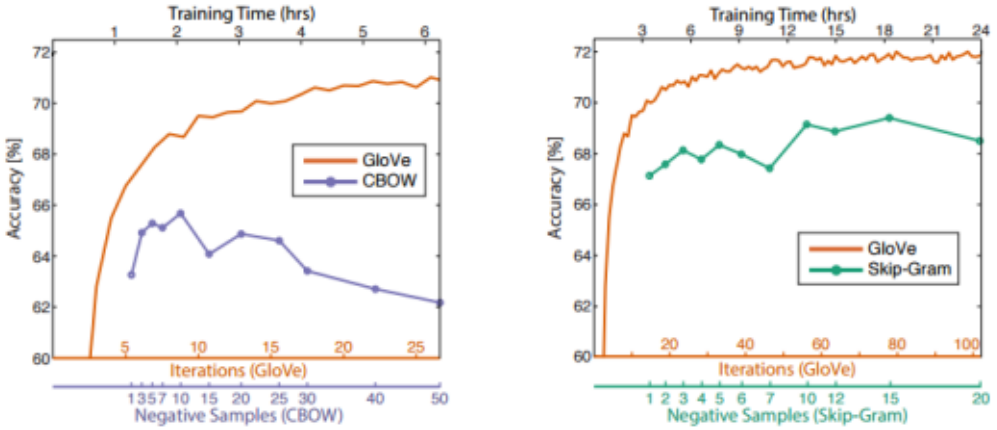


Figura 2.5: $f(x)$ *com* $\alpha = 3/4$

Os autores do GloVe também apresentaram um comparativo entre os dois modelos, como pode ser visto na figura 2.6.



(a) GloVe vs CBOW (b) GloVe vs Skip-Gram

Figura 2.6: *GloVe vs Word2Vec*

O GloVe foi o modelo escolhido para os testes comparativos realizados nos experimentos e descritos no capítulo 4. Como citado anteriormente, ele se baseia nas matrizes de frequência global dos termos para extrair informações de contexto, e essa também é uma tarefa definida nos algoritmos propostos pelo trabalho.

Random Indexing

Random Indexing é uma representação vetorial produzida a partir das estatísticas de coocorrência, com o intuito de diminuir a alta dimensionalidade encontrada em representações dessa natureza. Ao utilizarmos as coocorrências baseadas em documentos, por exemplo, as dimensões seriam iguais ao número de documentos existentes na coleção, e se usarmos as coocorrências baseadas nos termos, teríamos uma dimensionalidade

igual ao tamanho do vocabulário. Isso poderia nos levar a conclusão de que matrizes de coocorrência são inviáveis quando o vocabulário ou número de documentos é muito grande.

A principal desvantagem das matrizes de coocorrência é que normalmente a maioria dos elementos terá o valor zero, independentemente do tamanho da coleção de documentos ou do *corpus* utilizado. De acordo com Zipf [72], apenas uma pequena quantidade de palavras em uma determinada língua é uniformemente distribuída, sendo que a maioria das palavras aparecem apenas em um determinado número de contextos.

Alguns modelos foram propostos para contornar essa dificuldade com a esparsidade dos dados e a alta dimensionalidade das matrizes de coocorrência. Dentre esses modelos podemos destacar o LSA. O LSA utiliza técnicas de fatoração de matrizes para reduzir a dimensionalidade das matrizes de coocorrência e a esparsidade dos dados.

Contudo, alguns problemas podem ser observados na implementação de modelos como o LSA. Primeiramente, as matrizes de coocorrência devem ser construídas antes de serem reduzidas, isto é, o problema gerado pela alta dimensionalidade não é resolvido pelo método de redução de dimensionalidade empregado no modelo. Outro problema perceptível é que as técnicas de redução de matrizes, como o SVD, são computacionalmente intensas, consumindo muito tempo de processamento e memória. E finalmente, todo o processo deve ser refeito, do zero, caso seja necessário construir uma nova representação de um termo adicionado ao *corpus* ou um novo documento adicionado à coleção.

Com o intuito de resolver os problemas associados aos modelos tradicionais, como o LSA, um modelo de representação vetorial denominado *Random Indexing* foi proposto em [60], baseado no trabalho de [38]. O princípio fundamental da proposta é que os vetores sejam construídos a partir da coocorrência de termos num contexto, acumulando os valores destes vetores durante o processamento do *corpus*. Desta forma, não é necessário construir previamente a matriz de coocorrência e, consequentemente, remove-se a necessidade de se reduzi-la.

O processo pode ser resumido conforme mostra o Algoritmo 2.1.

Algoritmo 2.1: Random Indexing

Entrada: matriz $S[k]$ com os k vetores-semente
matriz $C[k]$ com os k vetores de contexto
lista T com o texto do corpus
quantidade N de elementos não nulos que serão inicializados
janela de contexto j
Saída: matriz $C[k]$

1 **para cada** $s \in S$ **faça**
2 inicializa o vetor s com $+1$ e -1 em $2 \times N$ posições aleatórias
3 **para cada** $c \in C$ **faça**
4 inicialize com 0s os vetores de contexto
5 **para cada** $t \in T$ **faça**
6 calcula a coocorrência do termo t em relação ao contexto
7 $\sum_{-j}^{j} c_t = c_t + s_{t+j}$

Para cada contexto encontrado, seja a partir de um documento ou de um termo, atribui-se uma representação vetorial esparsa, gerada aleatoriamente e única. A essa representação os autores deram o nome de *index vector*, mas outros autores, como [55], dão o nome de *seed vector*. Esses "vetores-semente" possuem esse nome porque são a base para a construção dos vetores de contexto resultantes do processamento.

Os elementos dos vetores-semente podem assumir apenas um de três valores distintos: 0, $+1$ e -1. A quantidade de $+1$s e -1s deve ser igual e pequena, se comparada à dimensão do vetor. Normalmente essa quantidade gira em torno de 10 elementos em um vetor com dimensão $d = 1024$. A posição desses elementos deve ser escolhida aleatoriamente, e deve-se garantir que esses valores estejam em posições diferentes. Por exemplo:

$$[1,0,0,0,-1,0,-1,1,0,\ldots,0,1,0,-1]$$

e

$$[0,-1,0,1,0,0,0,1,0,\ldots,-1,0,0,0]$$

Ao final do processamento realizado no Algoritmo 2.1, a matriz C contém vetores que representam os termos únicos encontrados dentro do *corpus*. Esses vetores possuem a vantagem de apresentarem uma dimensionalidade reduzida quando comparados com os vetores *one-hot* para o mesmo *corpus*, e permitirem que novos termos sejam adi-

cionados e vetores existentes sejam atualizados, simplesmente porque são compostos pela soma dos vetores que compõem suas vizinhanças.

Modelo Neural Clássico

Bengio et al. [8] propuseram um modelo neural de linguagem (Figura 2.7), consistindo-se de três camadas ocultas (*hidden layers*) numa rede neural *feed-forward*, capaz de predizer a próxima palavra dentro de uma sequência. Este foi um dos primeiros modelos a apresentar o conceito conhecido atualmente como *word embedding*. Esta arquitetura foi usada como protótipo para as pesquisas posteriores, que a melhoraram gradualmente. As camadas que formam o modelo são encontradas na maioria dos modelos neurais e de *word embeddings* atuais usados em NLP. Como pode ser observado na Figura 2.7, estas camadas são:

- *Embedding layer* – a camada de entrada que produz *word embeddings* através da multiplicação de um vetor de índices pela matriz de *word embedding*. Esta camada é representada na figura pelos elementos inferiores (retângulos menores), onde os vetores são recebidos pela rede;
- *Intermediate layer* – constitui uma ou mais camadas que transformam a entrada numa representação intermediária, como, por exemplo, uma camada *fully connected* que aplica alguma função de não-linearidade. Esta camada é representada na figura pelo elemento intermediário, com o título `tanh`;
- *Softmax layer* – é a camada de saída, responsável por produzir a distribuição probabilística das palavras em V, representada na figura pela camada superior, de mesmo nome.

Uma característica marcante deste trabalho foi a identificação de dois problemas que afetam até hoje os modelos considerados “estado da arte”:

1. Foi demonstrado que a(s) camada(s) intermediária(s) poderia(m) ser substituída(s) por uma rede LSTM (*long short-term memory*).
2. A camada *softmax* final tem um impacto maior na execução do treinamento, visto que o custo computacional do *softmax* é proporcional ao número de palavras em V, que, dependendo da base de dados utilizada, terá milhões de palavras. Encontrar novas e melhores formas de calcular essa distribuição probabilística é uma das abordagens adotadas atualmente por pesquisadores para aumentar a performance do *word2vec* [24] [62].

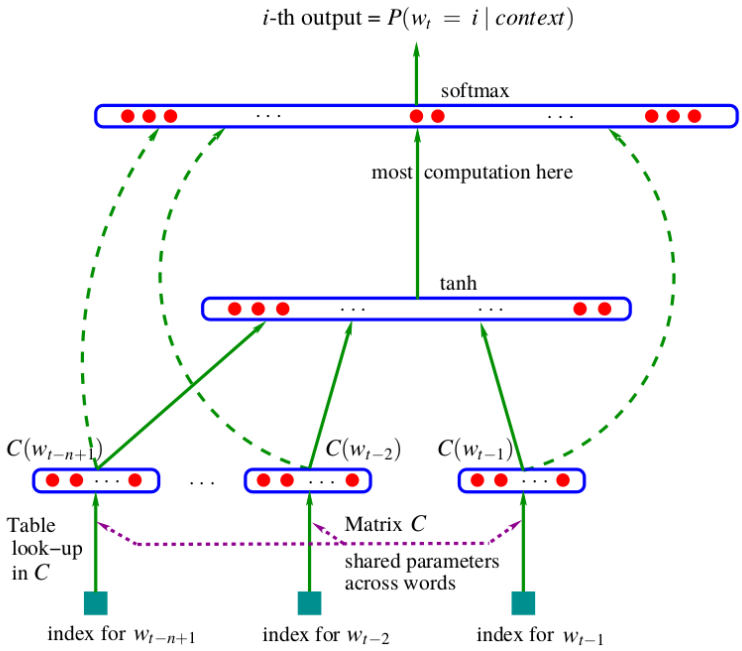


Figura 2.7: *Arquitetura de uma rede neural [8]*

Word2Vec

Mikolov et al. [45] propuseram duas arquiteturas para aprendizado de *word embeddings* que exigem menos esforço computacional que os modelos anteriores. Em [46] eles aperfeiçoaram estas arquiteturas com a adição de estratégias para melhorar a velocidade de treinamento e a acurácia. Assim, essas arquiteturas oferecem um benefício extra ao permitir que o modelo de linguagem absorva informações de contexto obtidas devido às características vetoriais do modelo de representação. Isto é, ao analisar a vizinhança de cada palavra é possível absorver características semânticas do texto dentro dos vetores que representam cada palavra. Essas arquiteturas são detalhadas a seguir.

CBOW

Um modelo de linguagem faz suas predições pela análise das palavras que já foram analisadas. Tais modelos são avaliados pela sua capacidade de predizer a próxima palavra dentro de um *corpus*. Tal modelo que visa apenas produzir *word embeddings* não precisa se restringir à essa estratégia. Foi assim que [45] construíram seu primeiro modelo. Este modelo pega as n palavras anteriores e posteriores da palavra w_t para efetuar uma predição, conforme mostrado na figura 2.8. Este modelo foi denominado de *continuous bag-of-words* (CBOW) porque utiliza uma representação contínua e os princípios da *bag-of-words*, onde a ordem dos elementos não tem relevância no resultado.

O objetivo do CBOW aqui é um pouco diferente do modelo de linguagem. Ao invés de obter apenas as n palavras anteriores, o modelo captura uma "janela" de n

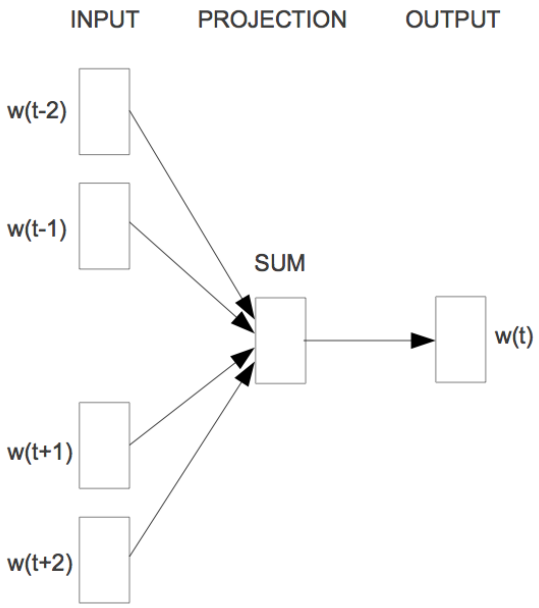


Figura 2.8: *Continuous Bag-of-Words [45]*

palavras ao redor do termo analisado w_t:

$$J_\theta = \frac{1}{T}\sum_{t=1}^{T} \log p(w_t | w_{t-n}, \ldots, w_{t-1}, w_{t+1}, \ldots, w_{t+n})$$

Skip-gram

O modelo CBOW apresentado anteriormente pode ser visto como um modelo pré-cognitivo, isto é, baseado nas palavras da vizinhança determina-se a palavra ao centro. Esse grupo de pesquisadores propôs também outro modelo, que inverte essa abordagem, usando o termo central para determinar a sua vizinhança (figura 2.9). Este modelo, denominado de *skip-gram*, soma todas as probabilidades das n palavras na vizinhança (esquerda e direita) da palavra w_t para alcançar esse objetivo.

$$J_\theta = \frac{1}{T}\sum_{t=1}^{T} \sum_{-n \leq j \leq n, \neq 0} \log p(w_{t+j} \,|\, w_t)$$

2.3 Algoritmos e Tabelas Hash

Esta seção apresenta conceitos básicos sobre *hash* e tabelas *hash*, e explica resumidamente alguns algoritmos que foram considerados para a realização desta pesquisa, e que levaram à construção da solução basilar deste trabalho.

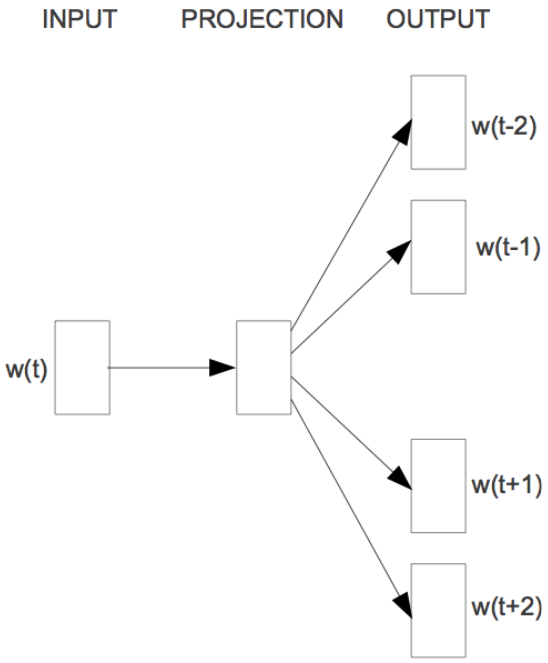


Figura 2.9: *Skip-gram [45]*

2.3.1 Algoritmos de Hash

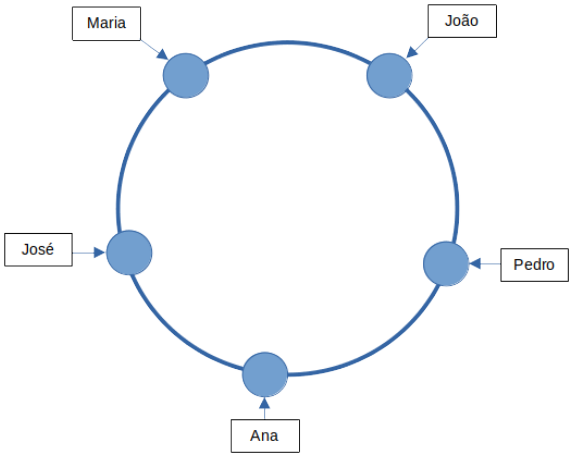
Algoritmos de *hash* permitem a transformação de um bloco de dados em uma representação resumida, normalmente de poucos *bytes* de comprimento. É comum encontrarmos o termo *digest*, que pode ser traduzido como "resumir", como sinônimo da operação realizada por meio desses algoritmos. A representação resumida dos dados recebe nomes diferentes, dependendo do objetivo da função de *hash*, por exemplo `chave`/`hash code` quando utilizamos o resultado da função para mapear um conjunto de dados em uma tabela; e *message digest* quando o valor obtido representa um resumo dos dados usados como entrada da função.

A Figura 2.10(a) mostra como uma função de *hash* mapeia um bloco de dados em um ponto sobre uma circunferência. Esta circunferência representa o "espaço de chaves" do algoritmo, isto é, todo o intervalo de valores que o resultado da função poderá assumir. Quando a função mapeia dois blocos de dados diferentes em um mesmo ponto dentro desta circunferência, temos o que é denominado "colisão" (Figura 2.10(b)).

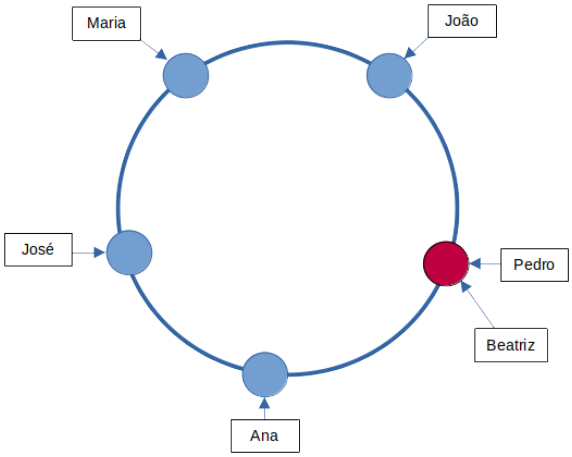
Existem duas classes de algoritmos de *hash*: os criptográficos e os não criptográficos. Os algoritmos de *hash* criptográficos fornecem, em teoria, a garantia de que uma vez resumidos os dados, não será possível determinar sua origem:

$$hash(x) = y \implies hash^{-1}(y) \neq x$$

Essa garantia é necessária devido aos cenários onde esse tipo de algoritmo é utilizado, como na autenticação de mensagens trocadas entre duas pessoas, por exemplo.



(a) *Nomes mapeados num espaço de hash*



(b) *Nomes mapeados num espaço de hash, mas houve colisão*

Figura 2.10: *Espaço de chaves de uma função hash*

Fonte: Chayner Cordeiro Barros

Alguns dos algoritmos criptográficos mais conhecidos são MD5[7], SHA-1[8], SHA-2[9], Whirlpool[10] e RIPEMD-160[11].

Os algoritmos não criptográficos, por sua vez, não oferecem tal garantia, sendo utilizados para mapear eficientemente um dado de tamanho variável em um bloco de bits de tamanho fixo. Neste tipo de *hash* não é incomum a ocorrência de colisões.

Para ambos os casos, as propriedades a seguir são relevantes e devem ser observadas na construção de um algoritmo de *hash*:

[7] MD5: https://tools.ietf.org/html/rfc1321
[8] SHA-1: https://tools.ietf.org/html/rfc3174
[9] SHA-2: https://tools.ietf.org/html/rfc4634
[10] Whirlpool: https://doi.org/10.1007/978-1-4419-5906-5_626
[11] RIPEMD-160: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.227.9050&rep=rep1&type=pdf

- Uniformidade — um algoritmo de hash deve produzir uma saída uniforme dentro do espaço de chaves. Para que isso aconteça, cada valor possível deve ter a mesma probabilidade de ocorrência. Isto é importante porque diminui as chances de **colisões**, que acontecem quando duas entradas diferentes produzem o mesmo resultado. Um algoritmo de *hash* cuja função alcance uniformidade absoluta é considerado *perfeito*. Não existe nenhum algoritmo conhecido com tal característica.
- Eficiência — devido à natureza das aplicações onde esses algoritmos são utilizados, espera-se que eles sejam eficientes, tanto no tempo quanto no espaço. Desta forma, espera-se que um algoritmo de *hash* consiga mapear um espaço de chaves com um número reduzido de instruções, e que o resultado (a representação do *hash*) seja pequeno o bastante para não ocupar tanto espaço quanto os dados originais.
- Determinístico — a implementação de um algoritmo de *hash* tem que ser uma função determinística, isto é, para a mesma entrada a função deverá produzir **sempre** o mesmo resultado.

Durante esta pesquisa, alguns algoritmos de *hash* não criptográficos foram analisados, levando em consideração as três características mencionadas anteriormente. O intuito era encontrar uma representação rápida e concisa o suficiente para representar termos na memória, sem colisão, de modo que os termos em si não precisassem ser manipulados diretamente.

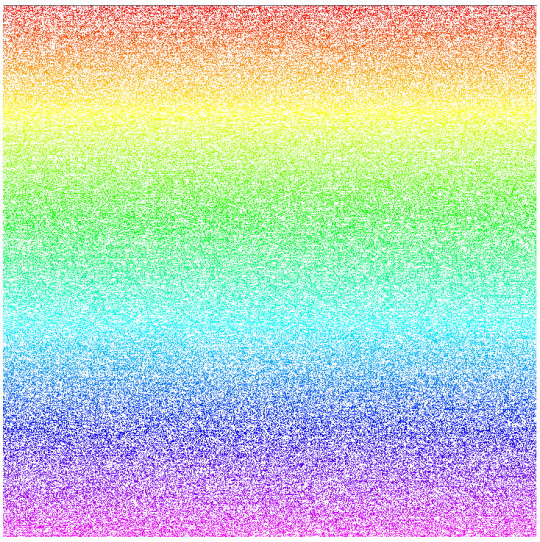
O gráfico mostrado na Figura 2.11 apresenta um espaço de chaves disperso quase uniformemente. Os pontos coloridos mostram as regiões no espaço de chaves utilizadas por um algoritmo, enquanto que os pontos brancos mostram regiões livres. A intensidade da cor mostra regiões mais ou menos saturadas com colisões. Se o algoritmo fosse *perfeito*, conforme explicado acima, não haveriam pontos brancos no gráfico, e as cores seriam mais claras.

As seções a seguir mostram alguns dos algoritmos não criptográficos mais conhecidos e utilizados. Foram utilizadas figuras contendo os gráficos de dispersão das chaves, que foram obtidas por uma análise realizada por Ian Boyd [29]. Nessa análise foram utilizadas três listas contendo 216.553 palavras no idioma inglês, os números de 1 a 216.553, e 216.553 GUIDs gerados aleatoriamente, em implementações de 32 bits dos algoritmos.

Destaca-se os algoritmos que foram analisados e testados, e que serão suscintamente apresentados a seguir:

- Lose-Lose
- DJB2
- DJB2a
- SDBM

Figura 2.11: *Uniformidade: o objetivo é distribuir uniformemente as palavras no espaço de chaves.*



Fonte: Ian Boyd, licenciado sob CC BY-NC-SA 4.0

- FNV1
- FNV1-A
- Murmur2

Lose-Lose

O algoritmo *Lose-Lose* pode ser considerado um dos algoritmos mais simples para transformação de um texto em um *hash code*.

Código 2.1 Função *lose-lose* implementada originalmente no livro *The C Programming Language*

```
1  #define HASHSIZE 100
2
3  hash(s) /* form hash value for string s */
4  char *si
5  {
6      int hashval;
7
8      for (hashval = 0; *s != '\0'; )
9          hashval += *s++;
10
11     return(hashval % HASHSIZE);
12 }
```

O trecho de código 2.1 apresenta a implementação original, publicada por Kernighan e Ritchie na primeira edição de seu famoso livro sobre a linguagem C em 1978 [39]. Conforme mencionado em [69], o algoritmo é um dos piores algoritmos de *hash* conhecidos.

SDBM

O algoritmo SDBM foi idealizado pelos autores em [18] e o código 2.2 apresenta uma implementação eficiente do algoritmo feita em ANSI-C e disponível no código-fonte do projeto GNU awk[12].

Código 2.2 Implementação do algoritmo *SDBM* em C disponível no código-fonte do programa `GNU awk`.

```
1  static unsigned long sdbm(str)
2  unsigned char *str;
3  {
4      unsigned long hash = 0;
5      int c;
6
7      while (c = *str++)
8          hash = c + (hash << 6) + (hash << 16) - hash;
9
10     return hash;
11 }
```

O SDBM é considerado um algoritmo com boa dispersão de chaves, com baixa probabilidade de colisão, mesmo com entradas muito grandes [63].

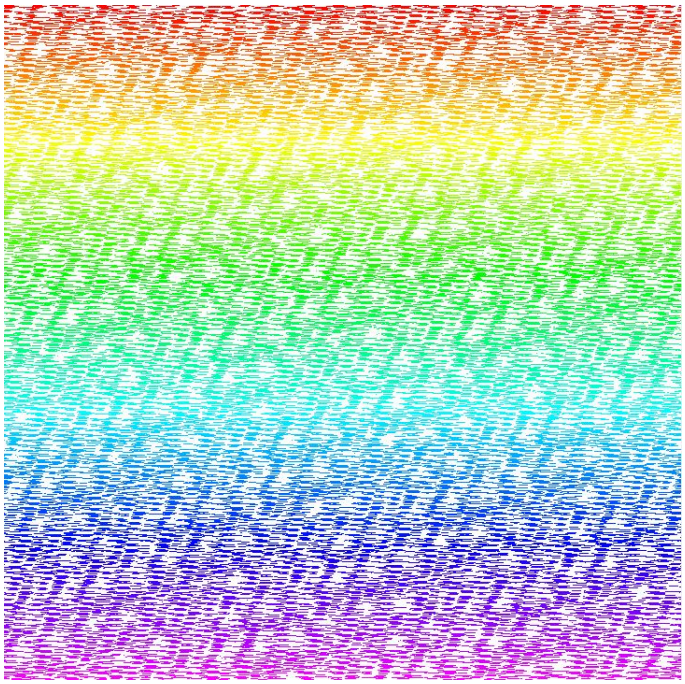
A Figura 2.12 mostra um gráfico de dispersão das chaves obtida por uma análise realizada por Ian Boyd [29], utilizando a função mostrada no código 2.2. Apesar de ser considerado um bom algoritmo, é possível observar a existência de regiões não mapeadas pela função de *hash*.

DJB2 / DJB2a

O algoritmo DJB2a é uma melhoria do algoritmo DJB2, descrito originalmente por Dan Bernstein no grupo de notícias `comp.lang.c` [69]. O algoritmo utiliza o número primo 33, escolhido empiricamente. A melhoria oferecida pelo DJB2a consiste apenas na troca da operação de adição (+) pelo operador lógico `XOR` na linha 7 do Código 2.3.

[12]Gawk: https://www.gnu.org/software/gawk/

Figura 2.12: *SDBM*



Fonte: Ian Boyd, licenciado sob CC BY-NC-SA 4.0

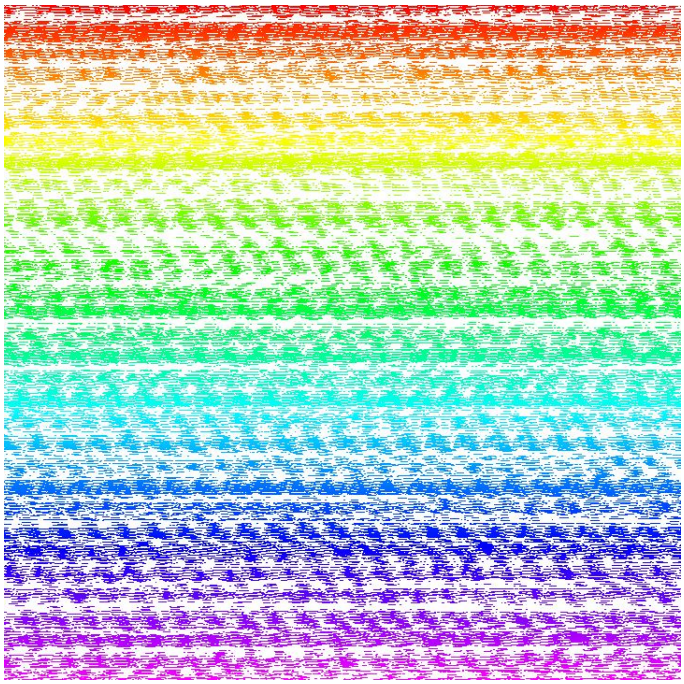
Código 2.3 Implementação do algoritmo *DJB2* em C.

```
1  unsigned long djb2 (unsigned char *str)
2  {
3      unsigned long hash = 5381;
4      int c;
5
6      while (c = *str++)
7          hash = ((hash << 5) + hash) + c; /* hash * 33 + c */
8
9      return hash;
10 }
```

A Figura 2.13 mostra o gráfico com a dispersão das chaves, também obtida por Ian Boyd [29].

É possível observar que o algoritmo, apesar de simples e eficiente, produz um espaço de chaves com diversas regiões não utilizadas, nos levando a conclusão de que há maior possibilidade de colisão entre as chaves geradas.

Figura 2.13: *DJB2a*



Fonte: Ian Boyd, licenciado sob CC BY-NC-SA 4.0

FNV-1/FNV-1a

O algoritmo de *hash* FNV-1 foi desenvolvido por Glenn Fowler, Landon Curt Noll e Kiem-Phong Vo (FNV), e submetido para padronização pela IETF (*Internet Engineering Task Force*) em setembro de 2011 [20]. Devido a sua eficiência e simplicidade, seu uso foi bastante difundido, sendo possível encontrar implementações do algoritmo em produtos como o *template* `hash_map` da STL do VC++ da Microsoft, em alguns dos serviços *web* do Twitter, na biblioteca de *runtime* do PHP 5.x, entre outros. O algoritmo e uma implementação de referência foram disponibilizados para domínio público.

A implementação do algoritmo permite que o tamanho das chaves seja variável, começando com 32 bits e indo até 1024 bits, sendo necessário apenas mudar o tipo de variável utilizada e os números primos utilizados para o cálculo do *hash* [42].

O código 2.4 apresenta uma implementação em C++ do algoritmo, usando os parâmetros de 64 bits. A variação entre o algoritmo FNV-1 e FNV-1a encontra-se na ordem como as operações de `XOR` e multiplicação ocorrem (linhas 11 e 12). De acordo com Landon Curt Noll [42], a variante FNV-1a possui melhor dispersão para pequenos blocos de memória, geralmente menores que 4 bytes, e por isso costuma ser mais adotado. Ele também recomenda que a variante FNV-1a seja utilizada em detrimento da versão original, devido ao *feedback* da comunidade.

Código 2.4 Implementação do algoritmo *FNV-1* em C.

```
#include <cstdint>

#define FNV_OFFSET_BASIS UINT64_C(0xcbf29ce484222325)
#define FNV_PRIME        UINT64_C(1099511628211)

uint64_t fnv1a (const char *word)
{
    uint64_t hash = FNV_OFFSET_BASIS;
    uint8_t byte;

    while (byte = (uint8_t) *word++)
    {
        hash *= FNV_PRIME;
        hash ^= byte;
    }

    return hash;
}
```

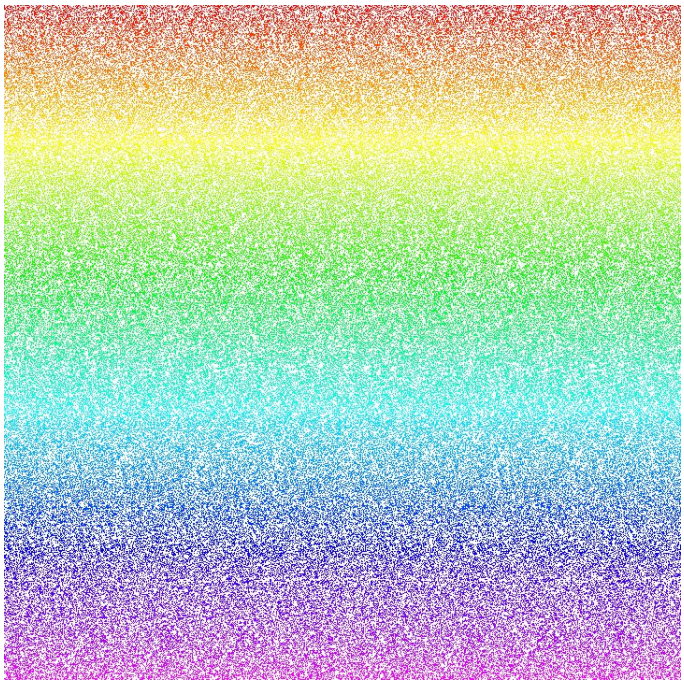
A Figura 2.14 apresenta o gráfico obtido por Ian Boyd [29], que nos mostra como o espaço de chaves no algoritmo FNV-1a é mais bem distribuído se comparado com os algoritmos apresentados anteriormente. De acordo com essa análise, o algoritmo conseguiu produzir *hashes* de todos os termos do conjunto de dados com apenas 4 colisões, utilizando uma implementação de 32 bits.

MurmurHash

O algoritmo MurmurHash foi projetado por Austin Appleby em 2008. Na sua versão atual (MurmurHash3) o algoritmo consegue computar *hash codes* de 32 e 128 bits de comprimento. O nome do algoritmo é derivado das operações que são realizadas em sequência para produzir o *hash*: **mu**ltiplicação e **r**otação.

Segundo o próprio autor, os *hash codes* produzidos pelas implementações nas arquiteturas x86 e x64 são diferentes, visto que os algoritmos foram otimizados para cada plataforma específica. Enquanto há um ganho de performance associado às otimizações inerentes da implementação para uma plataforma específica, a diferença das chaves produzidas reduz a compatibilidade entre programas que utilizam o algoritmo, princi-

Figura 2.14: *FNV-1a*



Fonte: Ian Boyd, licenciado sob CC BY-NC-SA 4.0

palmente quando os *hash codes* gerados podem ser armazenados e utilizados em uma implementação que está em outra plataforma.

A implementação do algoritmo está disponível gratuitamente na página do GitHub do autor[13], juntamente com a suite de testes de algoritmos de *hash* SMHasher.

A figura 2.15 mostra a dispersão das chaves na implementação de 32 bits. É possível observar que as chaves são mais bem distribuídas neste algoritmo, comparando-se com os mapas dos algoritmos apresentados anteriormente.

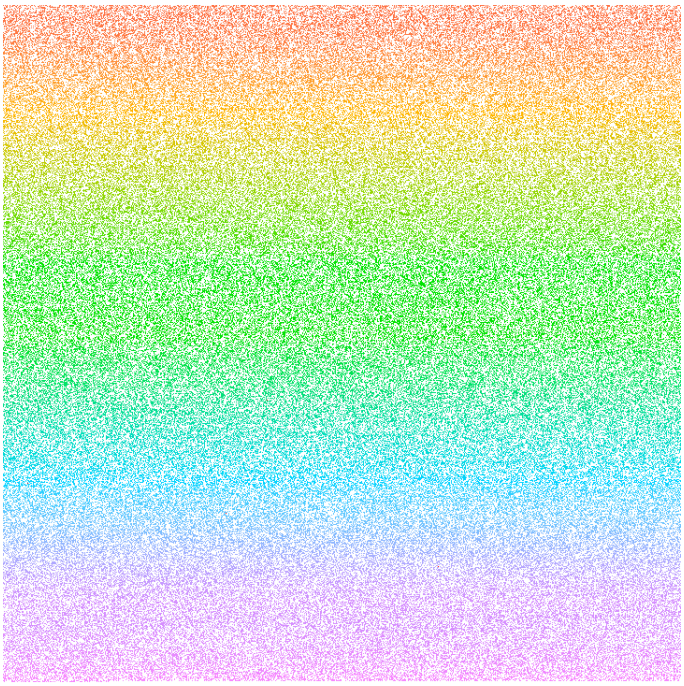
2.3.2 Tabelas Hash

Tabelas *hash* são estruturas de dados associativas que possibilitam o mapeamento entre um conjunto de dados e uma chave identificadora, que serve como índice na tabela. São eficientes do ponto de vista da complexidade, tanto no espaço quanto no tempo, possuindo complexidade $O(1)$ para a busca, remoção e inserção de elementos no melhor caso, e $O(n)$ no pior caso.

As tabelas *hash* recebem esse nome porque é necessário construir o índice de acesso aos elementos a partir da redução destes mesmos elementos. Essa redução, como discutida anteriormente, é realizada através de funções de *hash*. Ao processar os dados de entrada, a função de *hash* produzirá, preferencialmente, um valor único, que representa

[13] Austin Appleby - GitHub https://github.com/aappleby/smhasher

Figura 2.15: *MurmurHash*



Fonte: Ian Boyd, licenciado sob CC BY-NC-SA 4.0

apenas aquele elemento, sem ambiguidades. Contudo, funções de *hash* perfeitas não existem, e colisões são propensas a ocorrer.

O *hash* gerado pode então ser utilizado como índice para o elemento na tabela. Como, geralmente, a tabela *hash* possui um tamanho fixo e limitado, precisamos derivar o índice de acesso à tabela a partir do *hash* gerado. Comumente isto é feito através de uma operação de aritmética modular, onde o *hash* é dividido pelo tamanho da tabela e o resto da divisão é utilizado como índice:

$$\begin{aligned} \text{hash} &= f_{\text{hash}}(\text{dados}) \\ \text{pos} &= \text{hash} \bmod |\text{tabela}| \end{aligned}$$

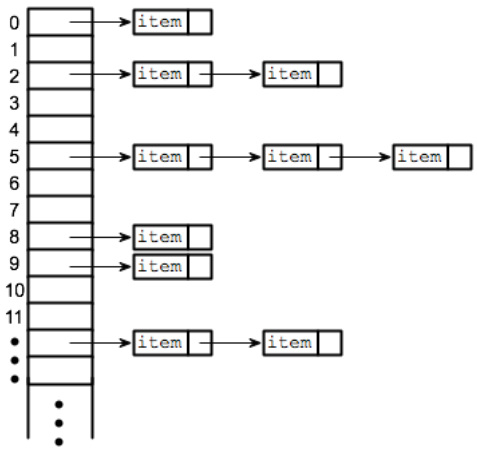
Desta forma, conseguimos "encaixar" o *hash* dentro dos limites disponíveis pela tabela. Contudo, ao realizarmos essa operação estamos reduzindo o espaço de chaves da função de *hash* original, o que aumentará a probabilidade de colisões.

Para contornar o problema causado pelas colisões, três técnicas podem ser utilizadas: encadeamento, endereçamento aberto e *hashing* duplo.

Encadeamento

O encadeamento é uma das formas mais comuns de resolução dos conflitos causados por colisão. Pode ser implementado utilizando-se uma lista encadeada (Figura 2.16) ou uma árvore de busca binária. Para se armazenar um elemento, insere-se este no final da lista encadeada, cujo ponteiro para o primeiro elemento é armazenado na tabela *hash*.

Figura 2.16: *Encadeamento como solução de colisões em tabelas hash*



Fonte: HackerEarth, licenciado sob CC BY-NC-SA 4.0

Por exemplo, na Figura 2.16, três elementos foram mapeados para a posição 5 (cinco). Nesse caso, um quarto elemento que tenha que ser armazenado na posição 5 terá que ser colocado ao final da lista. O mesmo princípio deve ser seguido para as buscas e remoções de elementos. Ao calcular o índice na tabela, percorre-se a lista apontada pelo *slot* até encontrar o elemento desejado.

Esta solução apresenta a complexidade $O(n)$ para o pior caso, onde todos os elementos são mapeados para a mesma posição, e todos devem ser armazenados no final da mesma lista encadeada.

Endereçamento Aberto

O endereçamento aberto é outra forma de resolução de colisão em tabelas *hash*. A diferença está na forma como os elementos conflitantes são armazenados. Ao invés de guardar os elementos em listas separadas, os elementos são armazenados na própria tabela, em posições subsequentes àquelas onde foram inicialmente mapeados. Isto é, se um elemento foi mapeado para a posição (*slot*) 5 da tabela, este será guardado neste espaço, caso esteja vazio. Se o *slot* já estiver ocupado por outro elemento, este deverá ser armazenado na posição 6, caso esteja vazio, e assim por diante.

Esta solução apresenta a complexidade $O(n)$ para o pior caso, onde todos os elementos são mapeados para a mesma posição, e todos devem ser armazenados na tabela, nas posições subsequentes. Neste caso, a tabela se comporta exatamente como um *array* unidimensional.

Hashing Duplo

Outra alternativa para a resolução de conflitos causados por colisão é o uso de *hashing* duplo, também conhecido como *2-choice hash*. Nesta abordagem a posição dentro da tabela é calculada com o uso de duas funções de *hash* $h_1(x)$ e $h_2(x)$.

Assim ao computar a posição (*slot*) onde o elemento será colocado, poderemos determinar qual das duas posições está com menos elementos, distribuindo mais uniformemente os elementos dentro da tabela.

2.4 Arquiteturas Paralelas e CUDA

2.4.1 Arquitetura Multicore

As arquiteturas paralelas são aquelas que permitem a execução simultânea de programas. O nível de paralelismo vai desde o nível de bits, instruções, dados até programas inteiros. O paralelismo está fortemente associado à concorrência, porém, é possível ter paralelismo sem concorrência, e concorrência sem paralelismo, como é o caso da execução concorrente de programas disputando o tempo de um único processador (*multitasking*) [67].

Esse nível de concorrência era necessário para atender à sempre crescente demanda por maior poder computacional e ao aumento da quantidade de programas disponíveis. Contudo, com os níveis de miniaturização na produção dos processadores chegando ao seu limite, outras soluções tiveram que ser elaboradas. A grande quantidade de transistores, associada com a alta frequência dos *clocks* internos dos processadores, causava um aquecimento excessivo, tornando o acréscimo de mais transistores, e o consequente aumento de performance por chip, impraticáveis. A alternativa idealizada foi a aplicação de paralelismo no projeto dos processadores, para atender às demandas da indústria.

Basicamente há duas abordagens aplicadas à produção de processadores para a obtenção de paralelismo: multiprocessamento e *multicore*. A princípio a indústria adotou o conceito de multiprocessamento, onde mais de um chip era instalado no mesmo sistema, permitindo a execução simétrica/assimétrica de programas. Contudo, esta alternativa possui as desvantagens de maior custo de produção, e menor aproveitamento do espaço, tanto no próprio chip, quanto nas placas-mãe. A segunda alternativa, desenvolvida mais

recentemente, coloca dentro do mesmo chip, dois ou mais núcleos (*cores*), sendo que cada núcleo individual possui todas as características de um processador comum. À essa arquitetura dá-se o nome de *multicore*.

As principais vantagens da arquitetura *multicore* incluem:

- Economia na produção dos chips;
- Possibilidade de produção de processadores com dezenas e até centenas de núcleos no mesmo chip;
- Conexões mais curtas entre os componentes internos, levando à uma maior performance e menor consumo de energia;

Atualmente existe também arquiteturas *multicore* que permitem a virtualização dos núcleos, que ao invés de executarem apenas uma única instrução por vez, podem dividir os recursos do núcleo entre duas ou mais *threads*. Esta tecnologia é conhecida como *Hyper-Threading* e foi criada pela Intel para os seus processadores Pentium 4, em 2002.

Para aproveitar os recursos disponíveis numa arquitetura *multicore*, utiliza-se bibliotecas especializadas para construção de programas concorrentes e paralelos. Dentre as diversas bibliotecas disponíveis, destaca-se:

- `pthreads`[14], que permite um nível de controle granular sobre a execução do código, ao custo de maior complexidade;
- `OpenMP`[15], que permite a paralelização de trechos de programas feitos em C, C++ e Fortran com o uso de apenas algumas anotações no código, ao custo de menor controle sobre como o código é paralelizado.

2.4.2 Arquitetura Manycore

Há poucas décadas os computadores não possuíam GPUs, sendo o processamento gráfico realizado totalmente pela CPU. Com a crescente demanda por gráficos melhores e mais rápidos, tornou-se necessária a construção de um componente especializado, capaz de realizar esse processamento cada vez mais complexo, liberando a CPU para trabalhos "mais importantes", como a execução dos programas em si.

Em 31 de agosto de 1999, a NVIDIA introduziu a sua primeira GPU para computadores domésticos, conhecida como GeForce 256 [49]. Ela era capaz de processar 10 milhões de polígonos por segundo, o que desafogou a CPU dessa carga gigantesca de processamento, considerando-se os processadores da época.

[14] pthreads(7) — Linux Manual Page: https://man7.org/linux/man-pages/man7/pthreads.7.html

[15] OpenMP Specification: https://www.openmp.org/specifications/

A adoção dessas placas dedicadas teve um papel importante na motivação por barramentos mais velozes (PCI[16], AGP[17], e atualmente PCIe[18]) e a adoção desses padrões nas placas-mãe. Também foram desenvolvidas APIs para processamento gráfico, como a OpenGL[19], a Direct3D[20] e mais recentemente a Vulkan[21].

A arquitetura das GPUs foi evoluindo de uma estrutura semelhante à das CPUs, com uma grande *cache*, ALU (unidade aritmética e lógica) e unidade de decodificação, para núcleos mais simples, contendo apenas elementos essenciais para a execução de instruções necessárias para o processamento individual dos vértices e pixels, assim como unidades dedicadas para etapas específicas do fluxo de renderização.

A ideia inicial era agregar vários desses núcleos simplificados, que permitiam o processamento de fragmentos em paralelo. Cada núcleo continha sua própria unidade de decodificação, uma ALU para execução da instrução, um banco de registradores e uma unidade controle do contexto da execução (Figura 2.17).

Desta forma, os núcleos poderiam ser agrupados em maiores quantidades, permitindo que vários fragmentos fossem processados simultaneamente, como pode ser visto na Figura 2.18.

Essa arquitetura apesar de permitir um alto paralelismo, quando comparado com as CPUs da época, ainda não era considerado ideal, visto que muitos elementos era redundantes e poderiam, por isso, ser removidos, liberando espaço para a adição de mais núcleos, e o consequente aumento de paralelismo.

Percebeu-se que o mesmo programa era executado em todos os núcleos. Por isso, uma nova abordagem foi adotada. Ao invés de duplicar núcleos completos, duplicava-se apenas as unidades responsáveis pela execução, compartilhando, portanto, a unidade de decodificação e parte do contexto de execução. Assim, uma mesma instrução é responsável por manipular diversos dados, em um modelo de execução conhecido como SIMD (*Single Instruction, Multiple Data*) (Figura 2.19). No modelo SIMD, a instrução informa o *array* contendo os dados que devem ser trabalhados simultaneamente pelas ALUs, eliminando a necessidade de busca e decodificação da mesma instrução e diminuindo-se a complexidade na execução dessas instruções. Com essa abordagem, mais núcleos puderam ser adicionados no mesmo chip, o que permitiu um aumento no paralelismo alcançado pelas GPUs.

[16] PCI (PCI SIG): https://pcisig.com/specifications

[17] AGP (Intel): https://web.archive.org/web/20150503042109/http://www.playtool.com/pages/agpcompat/agp10.pdf

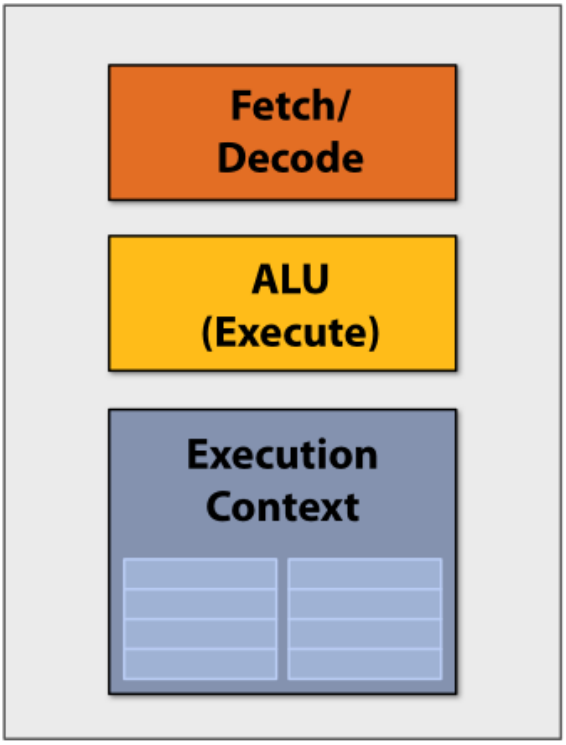
[18] PCI Express (Intel): https://www.intel.com/content/www/us/en/io/pci-express/pci-express-architecture-devnet-resources.html

[19] OpenGL (Khronos Group): https://www.khronos.org/opengl/

[20] Direct3D (Microsoft): https://docs.microsoft.com/en-us/windows/win32/direct3d

[21] Vulkan (Khronos Group): https://www.khronos.org/vulkan/

Figura 2.17: *Elementos encontrados dentro de um núcleo (*core*) de uma GPU*



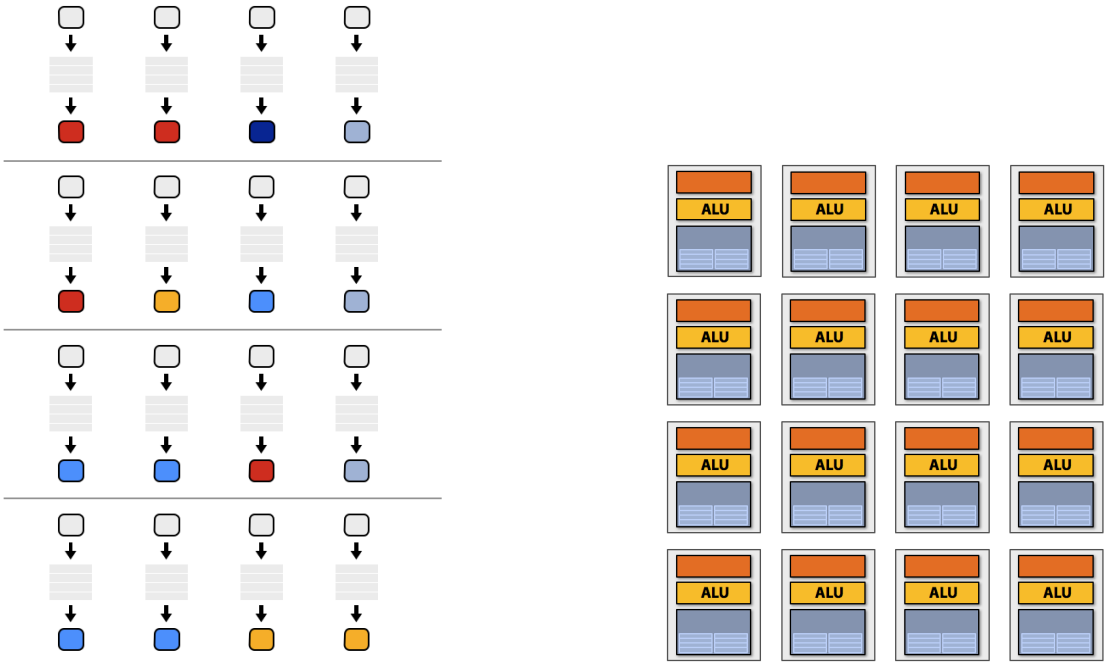
Fonte: Computer Graphics Course, Carnegie Mellon University

Com tanta capacidade de processamento paralelo disponível, alguns pesquisadores começaram a experimentar com a execução de programas não gráficos na GPU. Inicialmente, não havia nenhum suporte oficial dos fabricantes para o uso das GPUs com esse intuito, portanto os programas eram escritos como os *shaders*, usados para o processamentos dos fragmentos descritos anteriormente. A partir desses trabalhos um novo nicho surgiu, que aproveitava o poder computacional das GPUs para outras tarefas [50]. Denomina-se GPGPU (*General-Purpose computation on Graphics Processing Units*, ou Computação de Propósito Geral em GPUs) o paradigma que utiliza as unidades de processamento paralelas da GPU para a execução de programas. Duas tecnologias auxiliaram a difundir o uso das GPUs para a computação não gráfica: a OpenCL e a CUDA.

A OpenCL (*Open Computing Language*) é um padrão aberto para programação paralela, independente de plataforma. Foi lançada em 2008 pela Apple e é atualmente mantida pelo Khronos Group. A OpenCL fornece uma API que permite que programas sejam executados em múltiplos processadores simultaneamente em paralelo. Esses processadores podem ser CPUs, GPUs, DSPs (*Digital Signal Processors*) e FPGAs (*Field-Programmable Gate Arrays*). Com essa API, o processamento pode ser distribuído eficientemente entre esses processadores, o que pode aumentar a performance do programa.

A CUDA (*Compute Unified Device Architecture*), por sua vez, é uma plataforma

Figura 2.18: *16 núcleos de uma GPU, capazes de processar 16 fragmentos simultaneamente*



(a) *16 fragmentos* (b) *16 cores*

Fonte: Computer Graphics Course, Carnegie Mellon University

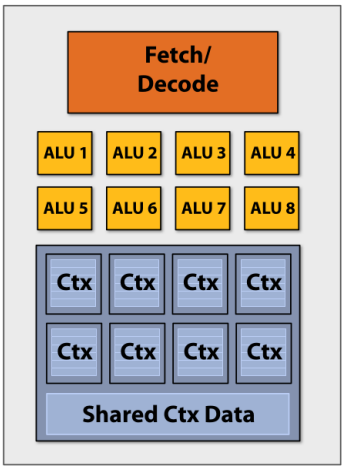
de computação paralela desevolvida pela NVIDIA e lançada em 2006. Ela permite que os programas utilizem tanto a CPU quanto a GPU. Devido ao suporte oferecido pela NVIDIA, tanto em capacitação quanto em bibliotecas, a CUDA é atualmente uma das arquiteturas GPGPU mais utilizadas, sendo uma tecnologia proprietária da NVIDIA, rodando apenas em seu hardware.

2.4.3 CUDA

Até 2006, os projetos de GPUs da NVIDIA estavam ligados aos estágios necessários para a renderização [21]. O modelo `GeForce 7900 GTX`, com o chip G71 era dividido em apenas 3 seções, que eram dedicadas ao processamento de vértices, fragmentos e a mesclagem desses últimos. Isso impedia que os projetistas determinassem os pontos de "gargalo" e os limitava às decisões impostas pelos fabricantes de APIs gráficas. Com o DirectX 10, outro elemento gráfico (*geometry shader*) se tornaria necessário nas placas, o que dificultaria ainda mais o trabalho dos projetistas em encontrar um equilíbrio entre os componentes da placa, principalmente porque não saberiam como esse novo elemento seria de fato utilizado e, como impactaria na performance do sistema como um todo.

Para resolver esse problema, a NVIDIA lançou o que denominou de arquitetura "unificada". No projeto do chip G80, não havia mais distinção entre as camadas de

Figura 2.19: *Núcleo SIMD de uma GPU)*



Fonte: Computer Graphics Course, Carnegie Mellon University

processamento. Toda a arquitetura anterior foi praticamente descartada, dando espaço para os denominados *Stream Multiprocessors* (SM).

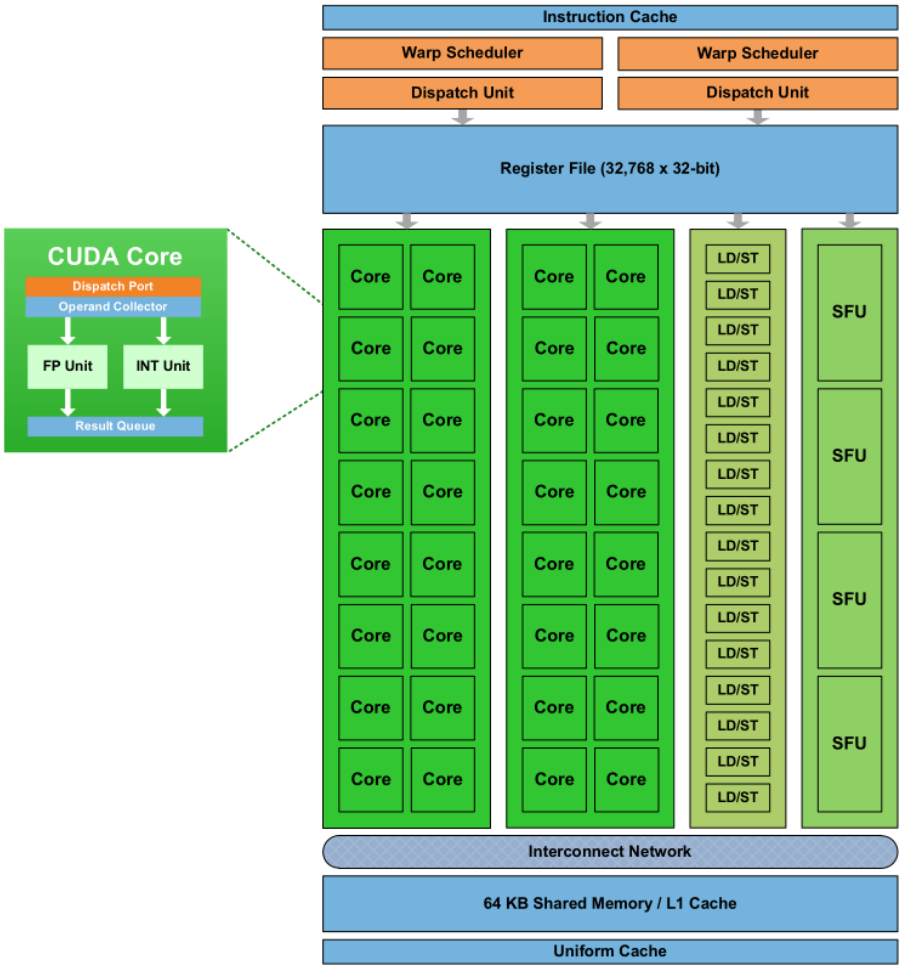
O SM da arquitetura Fermi (Figura 2.20) possui 32 núcleos, 16 unidades de *load/store*, 4 SFUs (unidades de funções especiais), um banco com 32K registradores de 32 bits, 64K de SRAM configurável, e uma unidade de controle. Como pode ser observado na miniatura da Figura 2.20, cada núcleo possui uma unidade de ponto flutuante e de números inteiros.

Dentro de um SM, os núcleos são divididos em dois blocos de execução, com 16 núcleos cada (Figura 2.21). Esses dois blocos, juntamente com as 16 unidades de *load/store* e os quatros SFUs formam 4 unidades de execução dentro de um único SM, capaz de disparar (*dispatch*) a cada ciclo de *clock* um total de 32 instruções, vindas de um ou mais *warps*. Para executar essas 32 instruções, um bloco de núcleos consome 2 ciclos. Caso seja uma instrução especial, as 32 instruções poderão ser disparadas em um único ciclo, mas serão necessários 8 ciclos para que sejam executadas pelos 4 SFUs disponíveis. Um SFU é uma unidade responsável por executar instruções matemáticas consideradas complexas, como `sqrt`, `sin`, `cos`.

De um ponto de vista macro, a arquitetura da CUDA é composta de vários componentes, divididos em camadas sobrepostas, interligando o hardware ao sistema operacional com o auxílio do *driver* da placa de vídeo e oferecendo uma interface de acesso dos aplicativos à GPU, através do *driver* da CUDA (Figura 2.22).

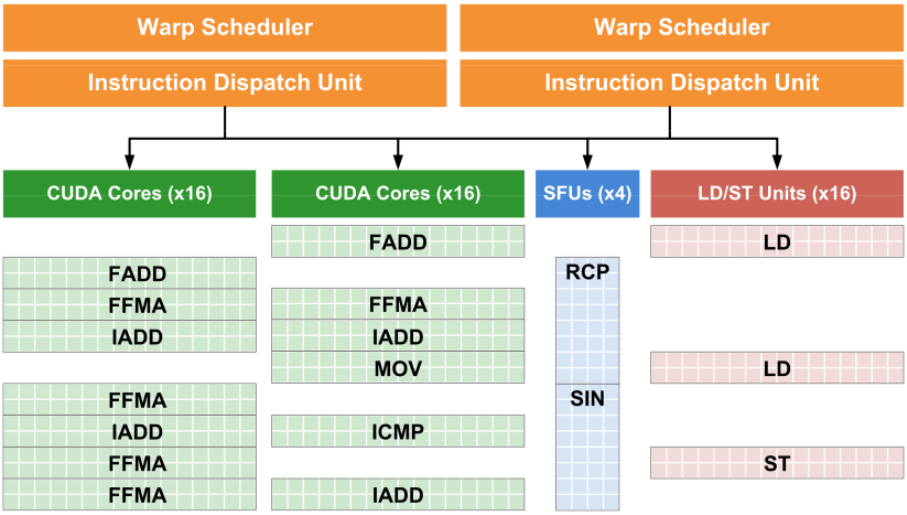
O paradigma de processamento da CUDA é formado por algumas abstrações importantes, que ajudam na compreensão da arquitetura e no desenvolvimento de aplicações que utilizam os recursos de processamento da GPU. Uma dessas abstrações é o *kernel*. Um *kernel* é a representação para a CUDA de um programa, e não deve ser confundido com o *kernel* dos sistemas operacionais. O *kernel* é de fato um procedimento contendo a

Figura 2.20: *Diagrama da arquitetura Fermi da NVIDIA*



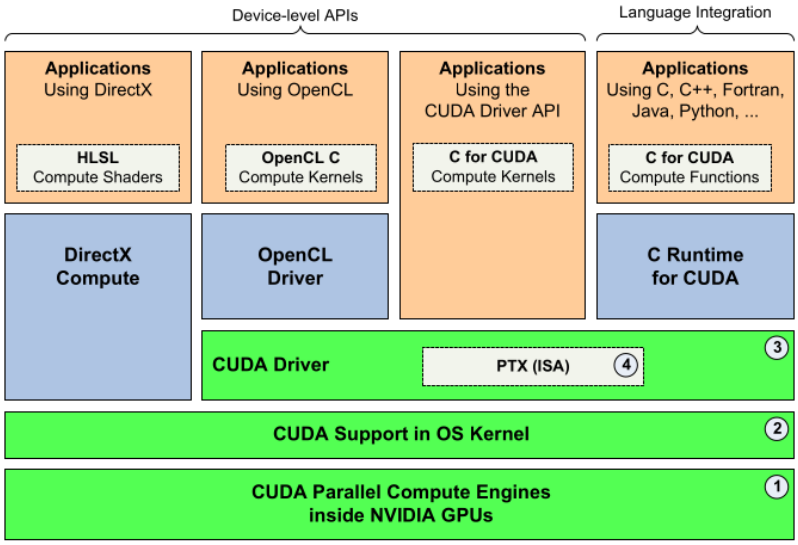
Fonte: NVIDIA's Fermi - the First Complete GPU Architecture

Figura 2.21: *Diagrama mostrando os blocos de execução em uma GPU NVIDIA da família Fermi e as unidades de controle destes blocos*



Fonte: NVIDIA's Fermi - the First Complete GPU Architecture

Figura 2.22: *Arquitetura CUDA*



Fonte: NVIDIA CUDA Architecture Overview

lógica que será executada em paralelo no dispositivo.

Para executar o *kernel*, uma ou mais *threads* serão criadas no dispositivo. As *threads* são agrupadas em blocos (*blocks*), que por sua vez são agrupados em grades (*grids*). Essa abstração permite a criação de 3 dimensões de controle, facilitando a manipulação de dados uni-, bi- e tridimensionais. Cada bloco contém um número específico de *threads*, definido pelo desenvolvedor e informadas ao *driver* em tempo de execução. Alguns fatores devem ser levados em consideração durante a escolha do número de *threads* por bloco, e do número de blocos por grade, como, por exemplo, a capacidade de ocupação do processador, quantidade de memória compartilhada disponível, e até mesmo a latência desejada. O número de *threads* por bloco é limitado em 1024 nos dispositivos atuais, mas esse número pode variar, sendo necessário que o programa consulte o *driver* em tempo de execução para determinar esse limite.

Esse modelo de execução, derivado do SIMD, é denominado como SIMT (*Single Instruction, Multiple Threads*), porque para cada instrução decodificada pela unidade de execução, várias *threads* serão executadas. Os dois modelos são semelhantes, variando na forma como os dados são vistos e tratados pelo código. No SIMD o dado é a abstração e o foco principal do controle do fluxo de execução. Por outro lado, no SIMT a *thread* é a abstração e o foco do controle é mantida sobre essa entidade.

Código 2.5 Trecho de código demonstrando uma estrutura condicional convencional

```
1  #include <nmmintrin.h>
2
3  void fn (void)
4  {
5      __m128 v = _mm_set_ps (1.0f, 2.0f, 3.0f, 1.0f);
6      __m128 n = _mm_set1_ps (2.0f);
7
8      if (alguma_coisa == true)
9      {
10         // Todos os elementos do array 'v'
11         // terão seus valores alterados.
12         v = _mm_mul_ps (v, n);
13     }
14 }
```

O Código 2.5 apresenta um trecho de código utilizando a extensão SSE para o conjunto de instruções x86, suportado pelos principais processadores da Intel e AMD. Ele demonstra uma implementação seguindo o modelo SIMD, onde uma única instrução é capaz de processar vários dados simultaneamente. Na linha 5 e 6 vemos dois vetores declarados. Esses vetores fornecem a abstração de dados utilizada pelo modelo SIMD, isto é, cada instrução trabalha sobre um conjunto de dados "empacotados" como um vetor. O tipo de dados `m128` diz ao compilador que iremos trabalhar com um vetor composto por 4 valores em ponto flutuante de 32 bits, ocupando um total de 128 bits de armazenamento. O trecho de código demonstra uma das dificuldades de se trabalhar com o modelo de execução SIMD: caso algum dos valores do vetor não pudesse ser alterado, seria necessário "desempacotar" o vetor, ou mesmo criar uma máscara (nesse caso multiplicar o valor específico por `1.0f`).

Código 2.6 Trecho de código demonstrando uma estrutura condicional em CUDA

```
1  __global__
2  void fn (int v [], int n)
3  {
4      int pos = threadIdx.x;
5
6      if (alguma_coisa == true)
7      {
8          // Somente o elemento na posição 'pos' terá
9          // seu valor alterado.
10         v [pos] = n;
11     }
12 }
```

Enquanto isso, o Código 2.6 apresenta um trecho de código utilizando CUDA. Ele demonstra como o modelo SIMT oferece uma abstração mais intuitiva se comparada com o código anterior. Como cada *thread* possui independência lógica entre si, ao associar cada elemento do conjunto de dados a uma única *thread*, podemos trabalhar paralelamente os dados, assim como ocorre com o modelo SIMD, porém com a vantagem de que não é necessário realizar nenhuma operação extra, como desempacotamento de vetores ou criar uma máscara sobre os dados. Toda essa complexidade é gerenciada pela unidade de controle do SM.

Contudo, o SIMT também possui limitações. Conforme mostrado na Figura 2.21, cada bloco de núcleos executa uma única instrução por vez. Diferentemente do que ocorre numa CPU, os núcleos são basicamente ALUs, que executam uma determinada operação, caso a *thread* dentro do *warp* esteja marcada para execução. Dentro do *warp* as *threads* que não entrarem dentro do desvio condicional serão mascaradas, isto é, receberão um bit informando que a execução da instrução para aquela *thread* deve ser ignorada. Ao mesmo tempo que essa abstração permite que o desenvolvimento seja mais natural, assemelhando-se com a programação realizada tradicionalmente nas CPUs, ela obriga que os desenvolvedores projetem seus algoritmos para evitar ao máximo *thread divergence*, que ocorre quando *threads* dentro do mesmo *warp* seguem fluxos de execução divergentes. A *thread divergence* diminui a performance geral do programa, ao sub-utilizar os recursos de processamento disponíveis.

Desta forma, podemos observar que a CPU e a GPU apresentam arquiteturas diferentes, ambas com suas vantagens e desvantagens. Essa diferença dificulta a mensuração da eficiência de algoritmos implementados nessas duas arquiteturas. A próxima seção in-

troduz um medida baseada no tempo, que visa simplificar a mensuração da performance de algoritmos executados em plataformas diferentes: o *speedup*.

2.4.4 Medidas de Performance

Speedup pode ser definido como uma medida de performance comparativa entre dois sistemas computacionais, executando o mesmo programa ou programas diferentes que resolvem o mesmo problema.

O conceito de *speedup* foi estabelecido por Gene Amdahl em seu trabalho [2], onde define a fórmula que, posteriormente, ficou conhecida como Amdahl's Law, que permite calcular o ganho de performance esperado quando um programa (ou parte deste) é paralelizado. A fórmula é apresentada na Equação 2-4.

$$S_{\text{ganho}}(s) = \frac{1}{(1-p) + \frac{p}{s}} \tag{2-4}$$

Assim:

- S_{ganho} é o ganho teórico obtido com a execução paralela de todo o programa;
- s é a parte do programa que se beneficiaria com o paralelismo; e
- p é o tempo de execução da parte do programa que se beneficiaria com o paralelismo.

A principal falha encontrada nesta teoria é que o problema analisado deve ter um tamanho fixo. Contudo, em problemas reais, à medida que mais recursos computacionais se tornam disponíveis, maiores se tornam os problemas tratados pelos programas, fazendo com que o tempo de processamento das partes paralelizáveis do programa aumentem mais do que as partes sequenciais. A Figura 2.23 demonstra esse problema. De acordo com a fórmula, mesmo com recursos computacionais elevados, e com praticamente 95% do programa executando em paralelo, obteríamos apenas um ganho de 20$\times$ em relação ao mesmo programa executando sequencialmente em sua totalidade.

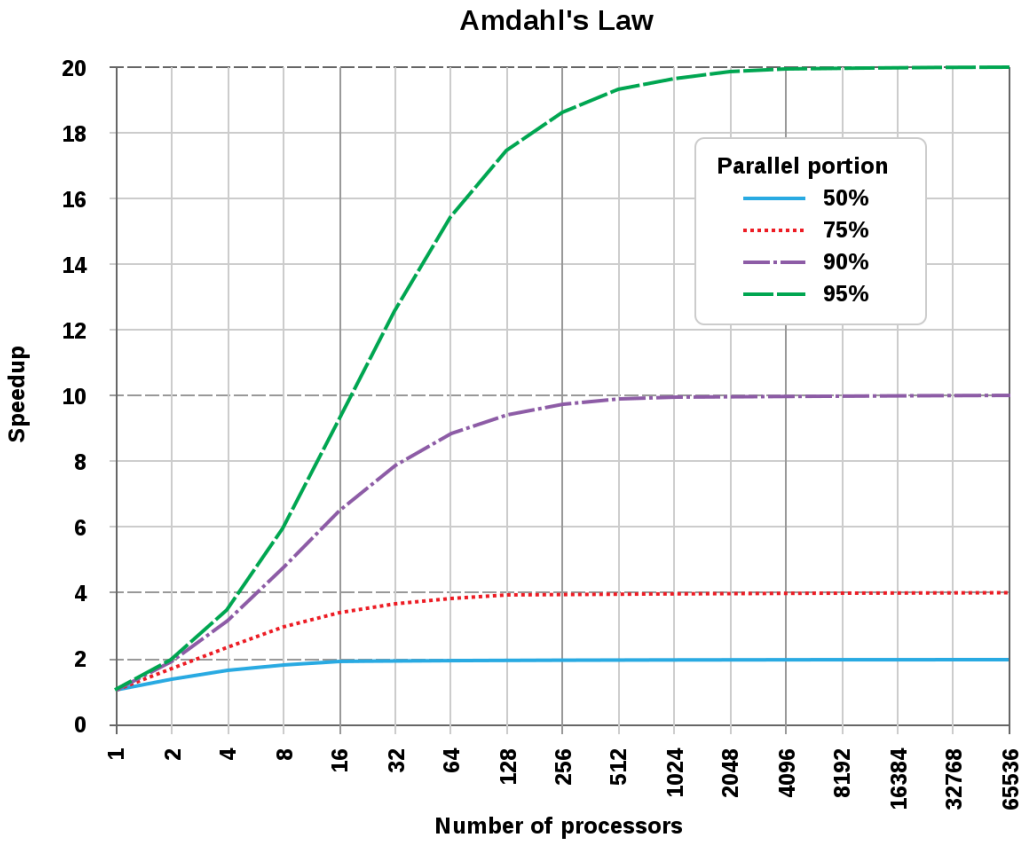
Posteriormente, John Gustafson redefiniu o conceito, a partir de seus estudos publicados em [26], onde reavalia a Lei de Amdahl. Ele estimou o ganho de performance S conforme descrito na Equação 2-5.

$$S = N + (1-N) \cdot s \tag{2-5}$$

Assim:

- S é o ganho de performance teórico obtido com a execução paralela de todo o programa;

Figura 2.23: *Lei de Amdahl*

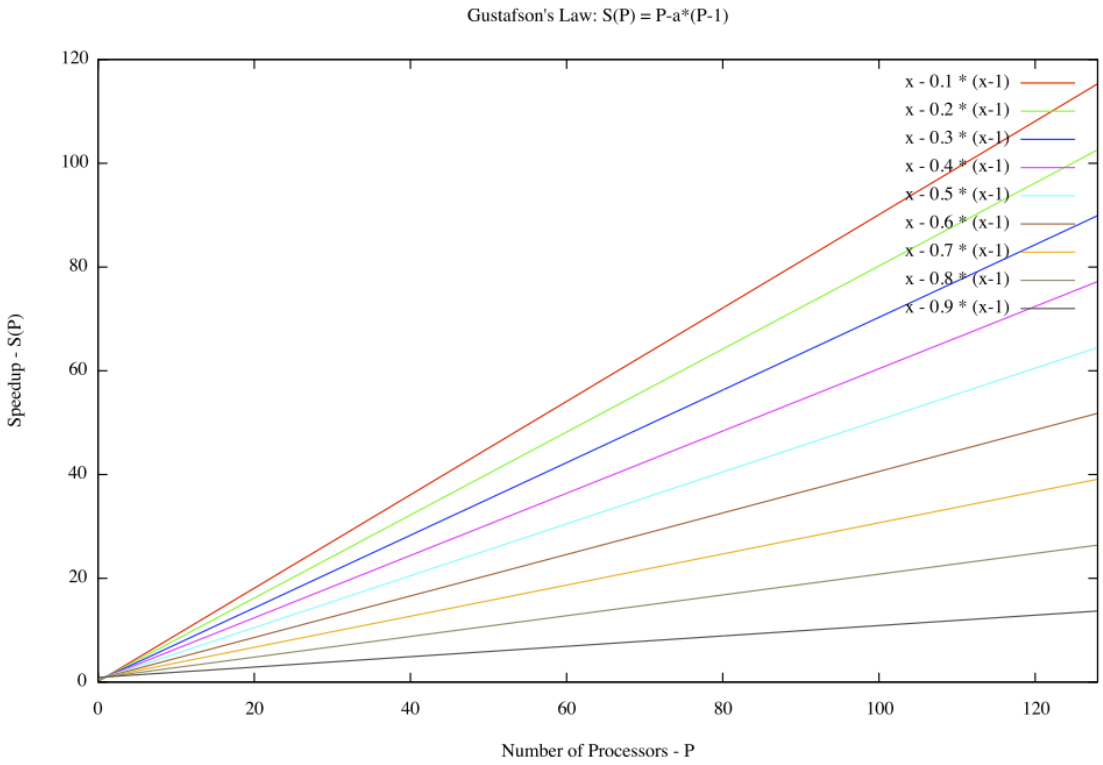


Fonte: Wikipedia, licenciado sob CC BY-NC-SA 4.0

- s é o ganho obtido com a execução da parte do programa que se beneficiaria com o paralelismo; e
- N é o número de processadores disponíveis.

A lei de Gustafson resolve o problema associado com a lei de Amdahl, que é baseada na ideia de que o problema tratado possui sempre um tamanho fixo, conforme dito anteriormente. Diferentemente de Amdahl, Gustafson propõe uma fórmula que assume a ideia que, à medida que os recursos aumentam, os problemas tratados também crescem, utilizando todos os recursos disponíveis, conforme demonstra a Figura 2.24.

Figura 2.24: *Lei de Gustafson*



Fonte: Wikipedia, licenciado sob CC BY-NC-SA 4.0

Como pode ser visto na Figura 2.24, à medida que mais processadores são colocados no sistema, maior o *speedup* obtido, visto que mais partes do programa podem ser executadas em paralelo.

Desta forma, por apresentar resultados mais precisos e condizentes com a realidade, as medições de ganho de performance utilizadas neste trabalho utilizaram a métrica estabelecida pela fórmula de Gustafson.

CAPÍTULO 3

Algoritmos Propostos

Neste capítulo são apresentados os algoritmos propostos por este trabalho.

Esses algoritmos foram projetados para demonstrar a viabilidade da realização das atividades de pré-processamento, construção da matriz de coocorrência e a geração de *embeddings* de modo eficiente na GPU. Para todos os casos, foram projetados algoritmos sequenciais que serviram de base para o aprimoramento apresentado pelas versões paralelas, tanto *multicore* quanto *manycore*.

O primeiro conjunto de algoritmos representa a solução idealizada para a construção de tabelas *hash* nas GPUs. Essa estrutura de dados é essencial para que os demais algoritmos e implementações se tornem possíveis. Os algoritmos estão descritos na Seção 3.1.

O próximo conjunto de algoritmos representa uma solução para permitir que o vocabulário e a matriz de coocorrência sejam extraídos de modo eficiente na GPU. Eles são descritos na Seção 3.2, referente ao pré-processamento. Durante este processamento são previstas diversas outras atividades, como a remoção de *stopwords*, por exemplo, conforme descrito na Seção 2.1.

O conjunto de algoritmos seguinte representa a solução idealizada para permitir que *embeddings* fossem construídos na GPU em passos simples e eficientes, utilizando a representação definida pelos algoritmos de pré-processamento. Eles são descritos na Seção 3.3.

3.1 Tabela Hash e Paralelismo

O algoritmo proposto é baseado numa tabela *hash* de endereçamento aberto. Isso significa que, em caso de colisão, os elemento serão inseridos nas próximas posições livres da tabela. Esse tratamento transforma a tabela num *array* no pior caso, mas permite que uma estrutura de dados de tamanho fixo seja alocada na memória da GPU. Apesar de haver o suporte a alocação dinâmica de memória na GPU, ela traz um *overhead*, que pode ser evitado com o uso do endereçamento aberto.

3.1.1 Algoritmo Sequencial para a Tabela Hash

O Algoritmo 3.1 apresenta a lógica utilizada para a construção da tabela *hash* utilizando apenas uma única *thread*. Este algoritmo forma a base para as melhorias desenvolvidas nos algoritmos paralelos.

O algoritmo recebe como entrada um número informando o tamanho da tabela em posições (*slots*). Isso permitirá a navegação pela tabela, caso a posição desejada já esteja ocupada.

Em seguida, calcula-se o *hash code* do elemento que se pretende inserir na tabela (Linha 1). A função `hash` pode ser qualquer função (ou grupo de funções) que faça uma operação de "resumo" do elemento, conforme apresentado na Seção 2.3. Após o cálculo do *hash code*, a posição dentro da tabela é calculada, obtendo-se o resto da divisão inteira entre o *hash code* e o tamanho da tabela (Linha 2).

Caso a posição desejada esteja ocupada por outro elemento, torna-se necessário varrer toda a tabela, procurando uma posição vazia. Essa varredura é feita nos dois laços seguintes. No primeiro (Linha 4) procura-se uma posição vazia, partindo da posição calculada e caminhando até o fim da tabela, definido pelo seu tamanho $|T|$. No sgundo laço (Linha 9) procura-se a partir do início da tabela, até se alcançar a posição exatamente anterior à posição calculada.

Por fim, ao encontrar a posição adequada, o elemento é inserido na tabela (Linha 14). Caso nenhuma posição esteja livre, um erro é retornado (Linha 16).

Algoritmo 3.1: Algoritmo de Inserção na Tabela *Hash* — Sequencial

Entrada: inteiro $|T|$ definindo o tamanho da tabela hash em posições
lista T com $|T|$ posições (*slots*)
item e que será inserido na tabela
Saída: tabela hash T com elemento inserido **ou** *erro*

```
1 h = hash(e)
2 pos = h  mod |T|
3 index = pos
4 enquanto index < |T| e não vazio(T_index) faça
5 |   incremente(index);
6 fim
```

Algoritmo 3.1: Algoritmo de Inserção na Tabela *Hash* — Sequencial (continuação)

```
7  se não vazio(T_index) então
8  |  index → 0
9  |  enquanto index < pos e não vazio(T_index) faça
10 |  |  incremente(index);
11 |  fim
12 fim
13 se vazio(T_index) então
14 |  T_index → e
15 senão
16 |  erro "Tabela cheia"
17 fim
```

3.1.2 Algoritmo Paralelo para a Tabela Hash

O Algoritmo 3.2 apresenta a lógica para a inserção na tabela *hash* utilizando-se processamento paralelo, podendo ser utilizado tanto na arquitetura *multicore* quanto na *manycore* (GPU). Assume-se que o algoritmo é executado simultaneamente por *N threads* distintas.

Para que a alta densidade de *threads* não impacte na performance do algoritmo, tornando-o um gargalo, operações atômicas são definidas. Essas operações devem ser garantidas pelo *hardware* onde o algoritmo for implementado. Uma operação atômica é definida como aquela que ocorre na sua totalidade, dentro de um intervalo de tempo fixo, onde há a garantia de que nenhuma outra atividade ocorrerá enquanto essa operação estiver sendo realizada. Por exemplo, uma operação atômica que copia um valor de um registrador para uma posição de memória deve garantir que, enquanto esse valor estiver sendo transferido do registrador para a memória, nenhuma outra *thread* ou processador acessará aquela mesma posição de memória. Assim, caso haja alguma unidade de execução tentando acessar o mesmo endereço de memória, esta deverá aguardar até a conclusão da operação.

Assume-se a existência dos seguintes mecanismos, com suporte à operações atômicas:

- **Comparação** — a comparação entre um elemento contido numa posição da tabela *hash* e um outro valor deve ser realizada de modo atômico. Isso garante que, mesmo que os dados tenham que ser movimentados entre memória e registrador (ou vice-versa), essa comparação resultará no resultado correto. Isso é necessário porque tais movimentações podem consumir vários ciclos de *clock*, o que poderia resultar na

comparação entre dois valores que não representam mais os valores contidos nas variáveis sendo comparadas;

- **Atribuição** — a atribuição de uma váriavel deve ser realizada de modo atômico. Isso garante que o valor sendo atribuído à variável não será sobreposto por outra atribuição ocorrendo simultaneamente.

Há 3 (três) operações atômicas necessárias para a descrição do Algoritmo 3.2: a comparação, definida pelo símbolo $=_{\text{atômico}}$; a atribuição, definida pelo símbolo $\leftarrow_{\text{atômico}}$; e a pseudofunção `vazio`, que verifica se uma posição da tabela encontra-se livre. Essas operações são definidas para fins algorítmicos, com o intuito de facilitar a compreensão, sendo que na implementação devem ocorrer simultaneamente, preferencialmente dentro de uma mesma operação, para garantir que entre uma comparação e uma atribuição, por exemplo, não hajam mudanças nos valores das variáveis.

Os conceitos empregados na operação de inserção servem para as demais operações, como remoção e leitura, mantendo-se toda a estrutura condicional utilizada.

Inicialmente calcula-se o *hash code* e a posição na tabela onde o elemento deverá ser inserido (Linhas 2 e 3).

A principal diferença, quando comparado com o algoritmo sequencial, está na necessidade de se garantir que apenas uma *thread* acesse determinada posição da tabela, num determinado instante, conforme explicado anteriormente.

Se o *hash* do valor contido na posição for diferente do *hash* calculado, devemos testar novamente a posição para averiguar se ela continua vazia, ou se ela já não foi ocupada pelo mesmo elemento, sendo inserido por outra *thread* durante o intervalo de tempo decorrido entre a comparação (Linha 6).

A condição testada na Linha 14 ocorrerá se após os testes anteriores, uma *thread* conseguir obter a propriedade da posição da tabela, e coincidentemente ela representar o mesmo elemento. Todas as estruturas condicionais precisam ser avaliadas, para que uma inserção duplicada não ocorra na tabela, removendo assim, sua confiabilidade.

Após se analisar todas as posições disponíveis a partir da posição computada, deve-se reiniciar a busca a partir do início da tabela até se alcançar a posição exatamente anterior à posição computada (Linha 24).

Algoritmo 3.2: Algoritmo de Inserção na Tabela *Hash* — Paralelo

Entrada: inteiro $|T|$ definindo o tamanho da tabela hash em posições
lista T com $|T|$ posições (*slots*)
item e que será inserido na tabela
Saída: valor *booleano* indicando se o elemento foi inserido na tabela

```
1  inserido ← falso
2  h ← hash(e)
3  pos ← h  mod |T|
4  index ← pos
5  enquanto index < |T| e inserido = falso faça
6      se hash(T_index) =_atômico h então
7          T_index ←_atômico e
8          inserido ← verdadeiro
9      senão
10         se vazio_atômico(T_index) então
11             T_index ←_atômico e
12             inserido ← verdadeiro
13         senão
14             se hash(T_index) =_atômico h então
15                 T_index ←_atômico e
16                 inserido ← verdadeiro
17             fim
18         fim
19     fim
20     incremente(index)
21 fim
22 se inserido = falso então
23     index ← 0
24     enquanto index < pos e inserido = falso faça
25         repita os comandos do laço anterior
26     fim
27 fim
28 retorne inserido
```

3.2 Aceleração do Pré-processamento e Matriz de Coocorrência

Para permitir que o algoritmo fosse escalável, sendo capaz de processar corpus de diferentes tamanhos e em vários computadores (ou GPUs) simultaneamente, determinou-se a necessidade de se definir uma forma eficiente e confiável de representar as palavras do vocabulário de modo uniforme.

Para a construção do vocabulário, uma das formas utilizadas é representar os termos encontrados no texto por uma posíção dentro de uma lista de termos. Assim, a primeira palavra encontrada terá ID $= 0$, a segunda ID $= 1$ e assim por diante. Desta forma, é indispensável manter uma tabela associativa entre as *IDs* e os termos do vocabulário. Como uma das formas para a construção do vocabulário é a contagem das frequências dos termos, então outra tabela associativa é criada, para associar a *ID* do termo com sua frequência.

Nas subseções seguintes apresentamos três algoritmos. O algoritmo sequencial 3.3 estabelece a referência que foi utilizada para a construção dos demais algoritmos. O Algoritmo 3.4 apresenta a versão paralela para arquiteturas multicore (CPU), enquanto que o Algoritmo 3.5 demonstra a versão paralela para arquiteturas manycore (GPU).

3.2.1 Algoritmo Sequencial do Pré-processamento e Matriz de Coocorrência

O objetivo do algoritmo é extrair o vocabulário V e a matriz de coocorrência M de uma coleção de documentos. O termo "documento" é uma definição genérica, que varia de acordo com a aplicação, sendo a implementação livre para definir os limites e tamanho de cada documento. Na implementação utilizada nos experimentos descritos na Seção 4.3, um documento é uma frase de comprimento variável delimitada pelo caractere `\n` (conhecido como `line feed`).

Como pode ser visto no Algoritmo 3.3, o vocabulário é uma estrutura de dados capaz de armazenar um *token*. O *token* mantém informações sobre cada termo encontrado durante o processamento de cada documento. No algoritmo são definidas como informações do *token* a *string* que o constitui (Linha 7) e um contador com a frequência com que esse termo foi encontrado em todos os documentos da coleção (Linha 8). A matriz de coocorrência opera de forma semelhante, com a diferença de que são armazenadas as frequências em que pares de termos coocorreram dentro de uma janela pré-definida (conforme descrito na Seção 2.2.1).

Os algoritmos sequencial e *multicore* consideram que a memória é infinita, por isso a inicialização das estruturas de dados são representadas com $[\infty]$. Na implementação

isso pode ser emulado com o uso de alocação dinâmica de memória.

Assim, o algoritmo começa com a inicialização das duas estruturas essenciais para o problema, o vocabulário V (Linha 1) e a matriz de co-ocorrência M (Linha 2), ambas implementadas com o uso de tabelas *hash*. A primeira contendo *tokens*, e a segunda pares (tuplas) de *tokens* e seus respectivos contadores. Então, uma iteração é realizada para processar cada arquivo de texto a, que contém os documentos da coleção A. Em cada arquivo podem existir um ou mais documentos d que, por sua vez, contêm um ou mais termos w. Para cada termo w encontrado, incrementa-se a quantidade de vezes em que este foi encontrado e acrescenta-se a *string* que o define ao conjunto V. Em seguida analisa-se a vizinhança do termo, incrementando-se a frequência dos pares encontrados e salvando-se essa informação na matriz M. O algoritmo funciona tanto em janelas simétricas quanto assimétricas, isto é, aquelas onde apenas uma das direções é considerada. A verificação de simetria é realizada na Linha 9. Desta forma, caso a janela seja simétrica, começamos a análise da vizinhança a partir dos elementos à esquerda da palavra w, ou em caso contrário, a partir da próxima palavra à direita. Assim, varre-se todas as palavras na vizinhança, no laço definido na Linha 14.

Algoritmo 3.3: Algoritmo Sequencial para o Pré-Processamento e Matriz de Coocorrência

Entrada: lista $A[k]$ com os k arquivos do corpus
inteiro j_n com o tamanho da janela de co-ocorrência
booleano j_{sim} indicando se a janela é simétrica
Saída: vocabulário, matriz de co-ocorrência

1 $V \leftarrow [\infty] : token$
2 $M \leftarrow [\infty] : tupla$

Algoritmo 3.3: Algoritmo Sequencial para o Pré-Processamento e Matriz de Coocorrência (continuação)

```
3  para cada a ∈ A faça
4      leia a para a memória
5      para cada d ∈ a faça
6          para cada w ∈ d faça
7              V ← w
8              incrementa o contador V_w
9              se j_sim então
10                 i ← pos(w) − j_n
11             senão
12                 i ← pos(w) + 1
13             fim
14             enquanto i < pos(w) + j_n e i < |d| faça
15                 se i ≠ pos(w) e i ≥ 0 então
16                     u ← d[i]
17                     incrementa o contador M_{w,u}
18                 fim
19             fim
20         fim
21     fim
22 fim
```

3.2.2 Algoritmo Multicore do Pré-processamento e Matriz de Coocorrência

A principal melhoria no algoritmo *multicore* (Algoritmo 3.4) é a distribuição dos arquivos de dados entre as unidades de processamento disponíveis (entidade denominada por *CPU* no algoritmo). Os arquivos são carregados para a memória e encaminhados para uma das unidades de processamento t, representando as *threads* (Linha 4). Se houver mais arquivos que as unidades de processamento disponíveis, os arquivos serão distribuídos uniformemente entre as unidades existentes. As estruturas de dados que armazenam o vocabulário V e a matriz de coocorrência M são replicadas para que cada unidade de processamento t tenha acesso exclusivo e não utilize *mutexes* ou outras formas de controle de concorrência, obtendo assim, um aumento de performance (Linhas 1 e 2).

Após o término do processamento paralelo, o vocabulário V (Linha 25) e a matriz de co-ocorrência M (Linha 28) de cada unidade de processamento t são mescladas,

formando respectivamente V_T e M_T, com as mesmas informações que seriam obtidas com o uso do algoritmo sequencial.

Algoritmo 3.4: Algoritmo de Pré-processamento — Multicore

Entrada: lista $A[k]$ com os nomes dos k arquivos do corpus
conjunto CPU com as unidades de processamento desejadas
inteiro j_n com o tamanho da janela de co-ocorrência
booleano j_{sim} indicando se a janela é simétrica
Saída: vocabulário $\rightarrow V_T$, matriz de co-ocorrência $\rightarrow M_T$

```
1  V ← [CPU, ∞] : token
2  M ← [CPU, ∞] : tupla
3  para cada a ∈ A faça
4      para cada t ∈ CPU faça em paralelo
5          leia a_t para a memória
6          para cada d ∈ a_t faça
7              para cada w ∈ d faça
8                  V_t ← w
9                  incrementa o contador V_{t,w}
10                 se j_sim então
11                     i ← pos(w) − j_n
12                 senão
13                     i ← pos(w) + 1
14                 fim
15                 enquanto i < pos(w) + j_n e i < |d| faça
16                     se i ≠ pos(w) e i ≥ 0 então
17                         u ← d[i]
18                         incrementa o contador M_{t,w,u}
19                     fim
20                 fim
21             fim
22         fim
23     fim
24 fim
```

Algoritmo 3.4: Algoritmo de Pré-processamento — Multicore (continuação)

```
25 para cada v ∈ V faça
26 |  V_T ← v
27 fim
28 para cada m ∈ M faça
29 |  M_T ← m
30 fim
```

3.2.3 Algoritmo Manycore do Pré-processamento e Matriz de Coocorrência

Devido às limitações de recursos arquiteturais existentes nas arquiteturas *manycore*, como a falta de um mecanismo de paginação de memória, por exemplo, o Algoritmo 3.5 foi projetado para permitir que o processamento seja escalável tanto no consumo de memória quanto na capacidade de processamento. É possível parametrizar esse consumo, a partir do estabelecimento de valores mínimos para os tamanhos do vocabulário (V_{size}), da matriz de co-ocorrência (M_{size}) e dos conjuntos de documentos (B_{size}). Os valores de V_{size} e M_{size} são definidos a partir dos valores de σ e λ, respectivamente, e limitados a até $1/3$ da capacidade de ϕ, que representa a quantidade de memória disponível no dispositivo *manycore*.

Os documentos possuem tamanho fixo, diferente do que ocorre nas versões sequencial e *multicore*, descritas anteriormente. Na implementação utilizada no experimento apresentado na seção 4.3.3, um documento é uma frase de comprimento fixo (1024 bytes) delimitada pelo caractere `\n` (*line feed*). Os documentos são processados em lotes de B_{size} documentos, submetidos para o grupo de execução S_i disponível.

Os grupos de execução S representam os conjuntos de *threads* t alocados para processar um conjunto de documentos B. O tamanho do conjunto S é definido a partir da quantidade de memória disponível em ϕ após a alocação de memória para o vocabulário V e a matriz de co-ocorrência M.

O processamento é feito de forma semelhante às versões anteriores, com os arquivos de texto sendo lidos para a memória e os documentos extraídos e agrupados no conjunto B. Assim que um conjunto B estiver cheio, este é submetido para execução no dispositivo, que volta a carregar e submeter continuamente os documentos até que todos tenham sido processados.

Com o uso deste algoritmo é possível processar bases textuais maiores que a memória disponível no dispositivo *manycore*, permitindo uma maior escalabidade.

Algoritmo 3.5: Algoritmo de Pré-processamento — Manycore

Entrada: lista $A[k]$ com os nomes dos k arquivos do corpus
inteiro j_n com o tamanho da janela de co-ocorrência
booleano j_{sim} indicando se a janela é simétrica
inteiro B_{size} com o tamanho do espaço reservado para documentos
inteiro σ com o tamanho do espaço reservado para o vocabulário
inteiro λ com o tamanho do espaço reservado para a matriz
inteiro ϕ memória disponível na GPU
Saída: vocabulário $\rightarrow V$, matriz de co-ocorrência $\rightarrow M$

```
1  V_size ← min(σ, 1/3 φ)
2  M_size ← min(λ, 1/3 φ)
3  V ← [V_size] : token
4  M ← [M_size] : tupla
5  B ← [B_size]
6  S_size ← restante(φ)/B_size
7  S ← [S_size]
8  para cada a ∈ A faça
9      leia a para a memória
10     para cada d ∈ a faça
11         B ← d
12         se B está cheio então
13             S_i ← livre(S)
14             submete B para execução usando S_i
15             para cada t ∈ T[S_i] faça em paralelo
16                 para cada w ∈ d faça
17                     V ← w
18                     incrementa o contador V_w
19                     se j_sim então
20                         i ← pos(w) − j_n
21                     senão
22                         i ← pos(w) + 1
23                     enquanto i < pos(w) + j_n e i < |d| faça
24                         se i ≠ pos(w) e i ≥ 0 então
25                             u ← d[i]
26                             incrementa o contador M_{w,u}
```

3.3 Embeddings

Nesta seção descreve-se os algoritmos idealizados para a construção dos *embeddings* utilizando-se as técnicas definidas nas seções anteriores e o Algoritmo 3.6 para a construção da tabela *hash* para permitir o acesso rápido aos vetores densos computados.

A solução descrita baseia-se no *Random Indexing* (Algoritmo 2.1) que descreve uma forma de computar vetores densos a partir da análise dos termos que compõem a vizinha (contexto) de um termo central (Seção 2.2.1), e no Algoritmo 3.3, que descreve a solução proposta neste trabalho, para a leitura e pré-processamento de base de dados textuais.

3.3.1 Algoritmo Sequencial para Construção dos Embeddings

Nesta solução, os arquivos são lidos para a memória e processados à medida que as palavras são encontradas (Linha 11). De modo semelhante ao que ocorre no Algoritmo 3.3, o vocabulário é construído adicionando-se as palavras à tabela *hash* e incrementan-se o contador associado à palavra. Contudo, a construção do vocabulário é opcional, visto que a construção dos *embeddings* utiliza o *hash code* do termo para identificar os vetores de contexto e de semente (Linha 22). Desta forma, é possível reconstruir o identificador do vetor apenas calculando-se o *hash* do termo, não sendo necessário consultar o vocabulário para isso. Assim, o vocabulário deve ser computado apenas quando alguma atividade posterior necessitar das informações contidas nele.

Algoritmo 3.6: Algoritmo Sequencial para Construção de *Embeddings*

Entrada: lista $A[k]$ com os k arquivos do corpus
matriz $S[m]$ com os m vetores-semente
matriz $C[m]$ com os m vetores de contexto
tabela *hash* V para armazenar o vocabulário
quantidade N de elementos não nulos que serão inicializados
inteiro j_n com o tamanho da janela de coocorrência
booleano j_{sim} indicando se a janela é simétrica
Saída: vocabulário, vetores densos

1 $V \leftarrow [\infty] : token$

Algoritmo 3.6: Algoritmo Sequencial para Construção de *Embeddings* (continuação)

```
2  para cada s ∈ S faça
3  |  inicializa o vetor s com +1 e −1 em 2 × N posições aleatórias
4  fim
5  para cada c ∈ C faça
6  |  inicialize com 0s os vetores de contexto
7  fim
8  para cada a ∈ A faça
9  |  leia a para a memória
10 |  para cada d ∈ a faça
11 |  |  para cada w ∈ d faça
12 |  |  |  V ← w
13 |  |  |  incrementa o contador V_w
14 |  |  |  se j_sim então
15 |  |  |  |  i ← pos(w) − j_n
16 |  |  |  senão
17 |  |  |  |  i ← pos(w) + 1
18 |  |  |  fim
19 |  |  |  enquanto i < pos(w) + j_n e i < |d| faça
20 |  |  |  |  se i ≠ pos(w) e i ≥ 0 então
21 |  |  |  |  |  u ← d[i]
22 |  |  |  |  |  c_hash(w) = c_hash(w) + s_hash(u)
23 |  |  |  |  fim
24 |  |  |  fim
25 |  |  fim
26 |  fim
27 fim
```

3.3.2 Algoritmo Paralelo (Multicore e Manycore) para Construção dos Embeddings

O Algoritmo 3.7, por sua vez, apresenta a solução idealizada para a construção dos *embeddings* utilizando paralelismo. A tabela *hash* V utilizada para armazenar o vocabulário é de acesso global a todos os processadores (Linha 1). Nas Linhas 2 e 5 as matrizes são inicializadas em paralelo, sendo que as regiões de memória que as constituem são divididas entre os processadores disponíveis, eliminando-se a necessidade de qualquer

controle de concorrência. E cada termo de cada documento é processado por uma *thread* específica (Linha 11).

Algoritmo 3.7: Algoritmo Paralelo para Construção de *Embeddings*

Entrada: lista $A[k]$ com os k arquivos do corpus
matriz $S[m]$ com os m vetores-semente
matriz $C[m]$ com os m vetores de contexto
tabela *hash* V para armazenar o vocabulário
quantidade N de elementos não nulos que serão inicializados
inteiro j_n com o tamanho da janela de coocorrência
booleano j_{sim} indicando se a janela é simétrica
Saída: vocabulário, vetores densos

1 $V \leftarrow [\infty] : token$
2 **para cada** $s \in S$ **faça em paralelo**
3 inicializa o vetor s com $+1$ e -1 em $2 \times N$ posições aleatórias
4 **fim**
5 **para cada** $c \in C$ **faça em paralelo**
6 inicialize com 0s os vetores de contexto
7 **fim**
8 **para cada** $a \in A$ **faça**
9 leia a para a memória
10 **para cada** $d \in a$ **faça**
11 **para cada** $w \in d$ **faça em paralelo**
12 $V \leftarrow w$
13 incrementa o contador V_w
14 **se** j_{sim} **então**
15 $i \leftarrow \texttt{pos}(w) - j_n$
16 **senão**
17 $i \leftarrow \texttt{pos}(w) + 1$
18 **fim**
19 **enquanto** $i < \texttt{pos}(w) + j_n$ ***e*** $i < |d|$ **faça**
20 **se** $i \neq \texttt{pos}(w)$ ***e*** $i \geq 0$ **então**
21 $u \leftarrow d[i]$
22 $c_{\texttt{hash}(w)} = c_{\texttt{hash}(w)} + s_{\texttt{hash}(u)}$
23 **fim**
24 **fim**
25 **fim**
26 **fim**
27 **fim**

3.3.3 Proposta de Gerenciamento de Memória da GPU

Todos os algoritmos projetados foram testados e validados com experimentos, que serão apresentados no Capítulo 4. Com o auxílio destes experimentos foi possível melhorar alguns aspectos dos algoritmos, determinando-se pontos de contenção (gargalo), e pontos que poderiam obter maior ganho com o uso de paralelismo, por exemplo. Foi o caso apresentado no Algoritmo 3.8, onde é possível, com o uso de uma estrutura auxiliar, representada por U, sinalizar o mecanismo de execução, informando que não há mais espaço disponível na tabela *hash*. Assim, após realizar um *swap* da tabela *hash*, o processamento é reiniciado, e as *threads* t que não conseguiram executar anteriormente realizam suas operações (Linha 5), enquanto que as demais aguardam.

Com essa técnica é possível expandir virtualmente a memória disponível no dispositivo, desde que exista um armazenamento com capacidade superior, para onde as tabelas serão copiadas durante o *swapping* (Linha 11). Durante o processamento do laço principal (Linha 1), as tabelas submetidas para o espaço de swapping devem ser mescladas, produzindo-se uma tabela única H (Linha 15).

Algoritmo 3.8: Algoritmo Paralelo para Swapping da Tabela Hash

Entrada: elemento e sendo processado pelo algoritmo
conjunto T com todas as *threads* em execução
unidade de execução U
tabela *hash* H
espaço de armazenamento S
Saída: tabela *hash* H contendo todos os elementos.

```
1  para cada t ∈ T faça em paralelo
2      h ← hash(e)
3      index ← pos(h)
4      se não cheio(index) então
5          se não concluído(t) então
6              executa operação
7          fim
8      senão
9          sinalize U
10         paralise a execução
11         faça swap da tabela para S
12         reinicie a execução
13     fim
14 fim
```

Algoritmo 3.8: Algoritmo Paralelo para Swapping da Tabela Hash (continuação)

15 **para cada** *tabela* $\in S$ **faça em paralelo**
16 | mova os elementos de *tabela* para H
17 **fim**

CAPÍTULO 4

Experimentos

Conforme mencionado no capítulo 2, as atividades relacionadas ao pré-processamento e a extração de *features* são vitais para a construção dos algoritmos de representação vetorial de texto. E tais algoritmos, por sua vez, são de grande relevância nas áreas de classificação, agrupamento (*clustering*), recuperação de informação, processamento de linguagem natural, entre outras. Por esse motivo, foram realizados experimentos para determinar a viabilidade da execução dessas atividades utilizando-se a arquitetura da GPU como plataforma computacional, e um possível ganho em performance.

Experimentou-se com as atividades de pré-processamento, como a limpeza dos dados, a construção do vocabulário e a construção da matriz de co-ocorrência, assim como a criação de vetores densos utilizando a estrutura de dados proposta neste trabalho.

4.1 Metodologia

Os experimentos descritos neste capítulo foram realizados para comprovar a eficácia dos algoritmos propostos. Todos os experimentos foram executados 10 vezes consecutivas, para cada arquitetura correspondente, onde a média do tempo de execução foi extraída. A medição foi feita através da ferramenta `time`[1] disponível nos sistemas Unix/Linux.

Os programas produzidos foram executados em 3 computadores com configurações diferentes. Isso foi necessário devido às limitações de *hardware* existentes no computador principal (denominado **exp**), onde os experimentos foram desenvolvidos. O computador denomiado **ufg** foi utilizado para a construção dos *embeddings* com o algoritmo *Word2Vec*, enquanto que a máquina denominada **vm** foi utilizada para a construção dos *embeddings* com o algoritmo *GloVe*. Como as configurações dos outros computadores

[1] Apesar de existirem métodos de mensuração mais eficientes e com maior precisão, como `clock_gettime` ou a instrução `RDTSC`, esses métodos não estão disponíveis em todas as arquiteturas e plataformas.

são diferentes, as medições realizadas nessas máquinas não se basearam em tempo, para não comprometer a veracidade das informações apresentadas neste capítulo.

O computador **exp** possui as seguintes configurações:

- **Processador**: AMD A10-7850K, 4 cores @ 3.7 GHz
- **Memória**: 16 GB DDR3
- **GPU**:
  - **Modelo**: NVIDIA GTX 1050 TI
  - **Memória**: 4 GB DDR5 @ 3500 MHz
  - **Memory bus**: 128-bit
  - **CUDA cores**: 768 @ 1455 MHz
  - **SM**: 6
  - **Copy engines**: 2

O computador **ufg** possui as seguintes configurações:

- **Processador**: Intel Xeon E2620, 12 cores (24 threads) @ 2.0 GHz
- **Memória**: 32 GB DDR3
- **GPU**:
  - **Modelo**: NVIDIA GTX TITAN
  - **Quantidade**: 4
  - **Memória**: 6 GB DDR5 @ 3500 MHz
  - **Memory bus**: 384-bit
  - **CUDA cores**: 2880 @ 1072 MHz
  - **SM**: 15
  - **Copy engines**: 1

O computador **vm** possui as seguintes configurações:

- **Processador**: Intel Xeon E5620, 4 cores (virtualizados) @ 2.4 GHz
- **Memória**: 88 GB DDR3
- **GPU**: N/A

4.2 Algoritmos de Hash

Entre os algoritmos analisados e apresentados na seção 2.3.1, dois se destacaram pela performance e confiabilidade: FNV-1a [20] e Murmur [5].

As figuras 2.14 e 2.15 mostram como as chaves são bem distribuídas nesses algoritmos. Os pontos brancos nessas imagens indicam regiões onde não há palavras mapeadas, e as regiões mais escuras denotam sobreposição (colisão). Baseado nos experimentos

realizados por Ian Boyd [29], foi possível observar quais algoritmos deveriam ser mais bem analisados e testados com *datasets* padronizados.

Desta forma, para melhor averiguar o comportamento desses algoritmos, realizou-se testes envolvendo a geração de *hashes* de todos os termos existentes no *corpus 1 Billion Word Benchmark* [11], juntamente com uma lista contendo aproximadamente 370.103 palavras da língua inglesa (em minúsculas)[2]. Foram analisados a quantidade de colisões encontradas e o tempo médio de execução de cada algoritmo. As implementações foram realizadas baseando-se nos códigos-fonte disponibilizados na Seção 2.3.1, utilizando-se C++ e uma arquitetura de 64 bits.

Algoritmo	370K (colisões)	1B (colisões)	Tempo médio (ns)
lose-lose	367.980	2.420.080	97
djb2	0	17.164	101
sdbm	0	0	106
fnv1-a	**0**	**0**	**92**
murmur3	13	693	175

Tabela 4.1: *Comparação das colisões e tempos de execução dos algoritmos de hash analisados durante a pesquisa.*

Os resultados obtidos com os algoritmos são apresentados na Tabela 4.1. A primeira e a segunda colunas apresentam as quantidades de colisões encontradas para as bases descritas anteriormente. A primeira relativa à lista de aproximadamente 370 mil palavras da língua inglesa e a segunda relativa à base *1 Billion Word Benchmark*. A terceira coluna mostra o tempo médio de execução do algoritmo em nanosegundos. Como pode ser observado, o menor tempo de execução obtido foi com o algoritmo *FNV1-A*, enquanto os algoritmos *Lose-Lose*, *DJB2* e *MurMur3* apresentaram colisões. Para determinar o melhor algoritmo de *hash*, que atendesse aos propósitos desta pesquisa, utilizou-se a seguinte metodologia:

- Baixa complexidade do algoritmo;
- Menor tamanho do código *assembly* gerado;
- Menor tamanho do *hash code*;
- Tempo médio baixo para se computar um *hash code*;
- Inexistência de colisões;
- Compatibilidade entre diferentes plataformas.

Os testes demonstraram que o algoritmo FNV1-A possui todas as características levantadas e consideradas essenciais para este trabalho. Não houveram colisões durante

[2]List Of English Words: https://github.com/dwyl/english-words

os testes, e não houveram colisões nos experimentos realizados posteriormente. O tempo médio necessário para se calcular um *hash* com o algoritmo FNV1-A foi de aproximadamente **92 ns**.

Apesar de distribuir mais homogeneamente os *hash codes* no espaço de chaves, conforme pode ser visualizado no gráfico 2.15, o MurMur possui apenas dois tamanhos de *hash* possíveis: 32 e 128 bits. Sendo necessário o dobro do espaço para armazenar os *hash codes* quando comparado com o FNV1-A. Outro fator limitante deste algoritmo é que não há compatibilidade entre os *hash codes* gerados por implementações em arquiteturas diferentes, o que impede seu uso numa solução que tem como um de seus objetivos exatamente a eficiência em arquiteturas diferentes. Ressalta-se que os resultados apresentados na Tabela 4.1 são relativos ao *hash code* de 32 bits.

4.3 Pré-processamento e Matriz de Coocorrência

O propósito destes experimentos iniciais foi testar a viabilidade de execução na GPU de tarefas tipicamente realizadas na CPU, e obter um *speed-up* em relação às implementações da CPU. Essas atividades incluem a limpeza da base de dados, com a remoção de palavras de baixa relevância (*stopwords*), e a dedução dos radicais das palavras; a extração do vocabulário; e a matriz de coocorrência.

Foram construídas 3 (três) versões do mesmo programa, cada uma seguindo um paradigma específico, mas implementando o mesmo algoritmo. A primeira versão implementada representa o paradigma sequencial tradicional, onde cada processo possui apenas uma *thread* de execução. A segunda implementação representa o paradigma *multithread*, onde cada processo possui *n threads* distintas, capazes de executar tarefas diferentes e independentes. E a terceira, representa o paradigma *manycore*, onde cada processo possui *n threads* distintas, executando na GPU ao invés da CPU tradicional.

A base de dados utilizada inicialmente nos testes foi a *20 Newsgroups Dataset*[43][34], sendo posteriormente substituída pela *1 Billion Word Benchmark*[11]. Com o intuito de demonstrar o potencial das soluções propostas, utilizou-se um vocabulário com todos os termos existentes, separados por espaços em branco, totalizando 2.438.611 termos únicos.

4.3.1 Implementação Sequencial

Para definir a arquitetura de *software* utilizada, implementa-se a versão de referência do Algoritmo 3.3, que utiliza apenas uma *thread* de execução. Ao final do experimento, essa versão é utilizada como *baseline* nas comparações dos tempos de execução.

(a) *Execução sequencial de programas e E/S* (b) *Execução paralela de programas e E/S*

Figura 4.1: *Sobreposição de E/S com a execução de programas*

Fonte: Chayner Cordeiro Barros

O seguinte fluxo de execução ocorre em todas as versões:

- Os arquivos de dados são mapeados diretamente para a memória do sistema;
- Os termos são mapeados e indexados pelo *hash*;
- A matriz de co-ocorrência é esparsa, implementada através de uma tabela associativa;
- Ao final da execução, o vocabulário obtido é ordenado e persistido em disco, para utilização em próximas etapas;
- A matriz é persistida em disco num arquivo de texto simples.

Na construção do experimento, o mapeamento dos arquivos em memória foi utilizado conjuntamente com uma dica (*advice*) para o gerenciador de memória do sistema operacional. Assim o sistema efetua a leitura dos arquivos assincronamente, na tentativa de liberar a *thread* de bloqueios causados pela leitura dos dados no disco (Figura 4.1(a)). Desta forma, o tempo gasto na leitura dos arquivos é sobreposto com tempo de execução do programa (Figura 4.1(b)).

A tabela 4.2 apresenta o tempo gasto na execução da versão sequencial.

	Tempo (s)
Vocabulário + Matriz	**245,397**
Total (+I/O)	**266,575**
Desvio-padrão	**2,164**

Tabela 4.2: *Tempo de execução da versão sequencial*

Como esta é a versão de referência, *baseline* para as demais, não há outros tempos de execução medidos que podem ser usados para comparação. Logo não há observações relevantes sobre os tempos de execução obtidos.

4.3.2 Implementação Multicore

Com o intuito de experimentar diversos cenários possíveis, foram implementadas 3 versões distintas do Algoritmo 3.4, com modificações na arquitetura do programa e nas

regras de bloqueio do acesso à memória. O paralelismo foi implementado utilizando-se a biblioteca `pthreads`[3].

Versão 1

Na primeira versão utilizou-se um número de *threads* igual ao número de arquivos de dados existentes no *corpus*. Observou-se que essa abordagem é proibitiva e se mostra ineficiente para *datasets* distribuídos em grandes quantidades de arquivos, como é o caso do *20 Newsgroups Dataset*, que possui aproximadamente 18.828 arquivos em uma de suas variantes; e em *datasets* que armazenam todos os dados em um único arquivo, como é o caso da *Wikipedia*.

Nessa versão, uma única tabela associativa é empregada no armazenamento do vocabulário, e seu acesso é controlado por um *mutex* global.

Houve apenas uma mensuração de desempenho, onde o tempo de execução obtido apenas para a construção do vocabulário foi superior a **4 horas**. Constatou-se que o péssimo desempenho estava associado à excessiva troca de contexto entre as 18.829 *threads* e ao tempo gasto por elas executando em *spin lock*, enquanto aguardavam pelo seu direito de acesso à tabela do vocabulário.

Versão 2

Na segunda versão utilizou-se uma abordagem diferente, associando o número de *threads* ao número de processadores disponíveis no sistema. Essa abordagem permitiu um ganho de desempenho por diminuir, principalmente, a contenção causada na versão anterior.

Novamente, devido ao desempenho insatisfatório desta versão, houve apenas uma mensuração do tempo de execução, onde o valor obtido apenas para a construção do vocabulário foi de aproximadamente **664,24 segundos**.

Versão 3

Analisando o comportamento percebido nas execuções das versões anteriores, observou-se a necessidade de diminuir a contenção no acesso à tabela que mantém o vocabulário. Desta forma, cada *thread* mantém uma tabela única para armazenar o vocabulário encontrado durante o processamento dos documentos, não havendo, portanto, *locking*. Ao final, uma única *thread* agrega todas as tabelas produzidas. Essa mesma abordagem foi

[3] pthreads(7) — Linux Manual Page: https://man7.org/linux/man-pages/man7/pthreads.7.html

aplicada ao processamento da matriz de co-ocorrência. Manteve-se a abordagem da versão anterior, onde o número de *threads* é igual ao número de processadores disponíveis no sistema.

A tabela 4.3 apresenta os tempos de execução desta versão. O *speedup* obtido é em relação à versão sequencial do mesmo algoritmo.

	Tempo (s)
Vocabulário + Matriz	**85,130**
Total (+I/O)	**108,613**
Desvio-padrão	**1,228**
Speed-up	$2,454\times$

Tabela 4.3: *Tempo de execução da versão paralela (CPU)*

Com essa versão foi possível observar os ganhos significativos de performance em relação à versão sequencial de referência.

4.3.3 Implementação Manycore

Assim como ocorreu com a versão paralela para a CPU, realizou-se a implementação de diversas versões para a GPU, com o intuito de testar condições distintas. Foram implementadas 5 versões diferentes do algoritmo 3.5.

Versão 1

A primeira versão é basicamente uma adaptação direta da versão de referência. Foi usada para facilitar o desenvolvimento das demais versões. Com isso foi possível observar que ela não é escalável para arquiteturas *manycore*. Nesta versão todos os arquivos de texto que compõem o corpus são copiados para a memória da GPU. Dois vetores são preenchidos com as posições de início e fim de cada documento. Os documentos são definidos como linhas terminadas com `\n` (*line feed*) e são as unidades básicas processadas pelas *threads* da GPU. O espaço restante da memória da GPU é alocado para as tabelas que guardam o vocabulário. Cada *thread* possui sua própria tabela de vocabulário, semelhante à versão paralela para a CPU apresentada anteriormente, e, por isso, não há *locking*.

Versão 2

Nesta segunda versão, um bloco de tamanho fixo é reservado para armazenar os documentos (conforme definido anteriormente). Há dois vetores que mantêm as posições

de início e fim de cada documento. O espaço de memória restante é usado para armazenar a tabela de vocabulário e matriz de co-ocorrência, uma para cada *thread*.

Uma grande desvantagem dessa abordagem é que pouco espaço na memória da GPU é reservada para os dados que serão processados de fato. Para um corpus grande como o *1 Billion Word Benchmark* com aproximadamente 3,9 GB, essas transferências entre as memórias da CPU e da GPU consomem mais tempo do que o tempo gasto pela GPU processando dados, o que torna essa versão bastante limitada e inviável.

	Tempo (s)
Vocabulário	**917,61**
Matriz	**1083,47**
Total (+I/O)	**2063,72**

Tabela 4.4: *Tempo de execução da versão paralela (GPU)[Versão 2]*

A tabela 4.4 mostra o tempo medido para a execução do algoritmo na base de dados *1 Billion Word Benchmark* no computador de teste. Devido ao péssimo desempenho, não foram realizadas outras mensurações.

Versão 3

Nesta versão, os documentos foram definidos como blocos de tamanho fixo com 256 bytes de comprimento, o que elimina a necessidade de se utilizar os vetores com as posições de início e fim dos documentos. A memória é dividida entre diversos grupos de processamento (*streams* na nomenclatura do CUDA), que podem executar de forma independente caso certas condições sejam alcançadas[4]. Cada *stream* possui um bloco de 64 MB para armazenar documentos, que são lidos até que esse espaço seja preenchido, e 262.144 tabelas para armazenar o vocabulário e a matriz de co-ocorrência, com 32 entradas cada. A quantidade de tabelas foi estabelecida a partir da relação entre o tamanho do documento (256 bytes) e a quantidade máxima de *threads* por bloco suportada atualmente pelo CUDA (1024 *threads*). A quantidade de entradas foi definida empiricamente. E a geração do vocabulário e a computação da matriz de co-ocorrência ocorrem concomitantemente.

A tabela 4.5 mostra o tempo medido para a execução do algoritmo na base de dados *1 Billion Word Benchmark* no computador de teste. Devido ao péssimo desempenho, não foram realizadas outras mensurações.

[4]As condições para execução concorrente entre *streams* está descrita na documentação do CUDA, acessível em: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#implicit-synchronization

	Tempo (s)
Vocabulário + Matriz	**375,99**
Total (+I/O)	**416,24**

Tabela 4.5: *Tempo de execução da versão paralela (GPU)[Versão 3]*

Versão 4

Esta versão diferenciou-se pelo uso de regiões de memória reservadas para os blocos de *threads* da GPU. Diferentemente do que ocorria nas versões anteriores, onde cada *thread* possuía suas próprias tabelas, nesta versão cada bloco mantém instâncias próprias do vocabulário e da matriz de co-ocorrência. Isto é, cada uma das 1024 *threads* do bloco armazenam seus resultados na mesma região de memória. A concorrência é controlada por operações atômicas, que garantem a integridade dos dados gravados nessas tabelas.

	Tempo (s)
Vocabulário + Matriz	**114,89**
Total (+I/O)	**119,63**

Tabela 4.6: *Tempo de execução da versão paralela (GPU)[Versão 4]*

A tabela 4.6 mostra o tempo medido para a execução do algoritmo na base de dados *1 Billion Word Benchmark* no computador de teste. Houve uma melhora considerável de performance em relação às versões anteriores. Com esta versão foi possível determinar alguns dos "gargalos" que degradavam a performance na versão anterior. Um deles é o pós-processamento das 262.144 tabelas produzidas na GPU a cada execução de um *stream*. Esse pós-processamento era realizada na CPU por uma única *thread*. Outro problema detectado foi a baixa utilização da GPU, devido a pouca quantidade de trabalho a ser realizado pelas *threads*, e o excesso de transferências entre as memórias da CPU e GPU, que consumia mais tempo do que o processamento propriamente dito.

Versão 5

Após a detecção das causas da degradação da performance, esta última versão foi desenvolvida, aplicando as melhorias que foram consideradas necessárias.

Nesta versão, a memória da GPU é dividida em dois segmentos. No primeiro deles é mantido a tabela do vocabulário e a matriz de co-ocorrência. Essa região de memória é inicializada com zeros e só é lida pela CPU no final do processamento. O segundo

segmento é dividido em blocos de documentos de tamanho fixo. Nos experimentos foram testados diversos tamanhos, entre 64 MB e 1024 MB. Devido à restrição na capacidade da memória da GPU utilizada para os testes, apenas 2 blocos de 768 MB foram utilizados e o restante da memória foi reservada para o primeiro segmento.

O tamanho dos documentos foram aumentados para 1024 bytes. Isso permite a tokenização de termos mais longos. Assim, para um bloco de 768 MB é possível armazenar 786.432 documentos, que são copiados assincronamente para a memória da GPU, enquanto um ou mais *streams* estão em execução.

Apenas ao final do processamento a tabela e a matriz existentes no primeiro segmento de memória da GPU são copiadas para a memória da CPU. O vocabulário é ordenado sequencialmente e armazenado num arquivo de texto simples. A matriz é armazenada em formato binário. Para diminuir o tempo necessário com I/O, as páginas de memória para onde é copiada a matriz de co-ocorrência são mapeadas diretamente no arquivo de saída.

	Tempo (s)
Vocabulário + Matriz	**15,210**
Total (+I/O)	**21,829**
Desvio-padrão	**0,312**
Streams	**2**
Speed-up	$12,212\times$

Tabela 4.7: *Tempo de execução da versão paralela (GPU)[Versão 5]*

A tabela 4.7 mostra o tempo medido para a execução do algoritmo na base de dados *1 Billion Word Benchmark* e o *speed-up* obtido em relação à versão sequencial.

4.3.4 Comparação de Desempenho com o GloVe

O próximo experimento realizado objetivou comparar a solução *manycore* desenvolvida com uma solução sequencial bem conhecida, que implementa um algoritmo de construção de *embeddings*. O algoritmo GloVe foi escolhido devido às semelhanças dos passos executados, conforme visto na seção 2.2.1. A comparação é realizada somente sobre os módulos da implementação sequencial do GloVe que fazem o pré-processamento e a construção da matriz de coocorrência[5].

Na fase de projeto do algoritmo, definiu-se alguns critérios que foram considerados importantes para um bom desempenho:

[5]Repositório do GitHub com o código-fonte do GloVe, disponibilizado pela Universidade de Stanford https://github.com/stanfordnlp/GloVe

- O processamento deve ser “online”, isto é, todas as etapas ocorrerão sempre que o processo for disparado;
- Deve ser escalável, sendo capaz de processar *corpus* de diferentes tamanhos e com características distintas;
- Redução do consumo de memória. Por exemplo, diminuição da quantidade de tabelas residentes em memória.

A tabela 4.8 apresenta os tempos obtidos com o GloVe, no processamento do vocabulário e na construção da matriz de co-ocorrência, com uma janela de tamanho 2, para o *1 Billion Word Benchmark*.

	Tempo (s)
Vocabulário + Matriz	**447,046**
Total (+I/O)	**459,563**
Desvio-padrão	**2,316**

Tabela 4.8: *Tempo de execução da versão sequencial do GloVe*

A tabela 4.9 mostra o *speed-up* obtido comparando-se os tempos apresentados na tabela 4.8 com os tempos apresentados na tabela 4.7.

	Tempo (s)
GloVe	**459,563**
GPUv5	**21,829**
Speed-up	$21,053\times$

Tabela 4.9: *Comparação entre o tempo de execução da versão sequencial do GloVe e a versão manycore GPUv5*

O ganho de performance se deve não apenas pela grande capacidade de processamento da GPU, mas também pela estratégia empregada nas transferências de dados entre as memórias da CPU/GPU e a tecnologia disponível nas GPU atuais. A GPU existente no equipamento de teste disponibiliza 2 *copy engines*, que são os mecanismos responsáveis por realizar essas transferências assíncronas enquanto a GPU está ocupada realizando algum tipo de processamento. A tabela 4.10 apresenta todos os tempos obtidos com a versão sequencial, *multicore* e *manycore*, e o tempo para o GloVe.

4.4 Embeddings

O último experimento consiste na construção de *embeddings* utilizando nossa proposta de implementação paralela do algoritmo *Random Indexing* (Seção 3.3). Para

	Sequencial	Multicore	Manycore	GloVe
Vocabulário + Matriz	**245,397** s	**85,130**	**15,210**	**447,046**
Total (+I/O)	**266,575** s	**108,613**	**21,829**	**459,563**
Desvio-Padrão	**2,164**	**1,228**	**0,312**	**2,316**
Streams	*N/A*	*N/A*	**2**	*N/A*
Speed-up × Seq.	*N/A*	**2,454**	**12,212**	-
Speed-up × GloVe	**1,724**	**4,232**	**21,053**	*N/A*

Tabela 4.10: *Tempos de execução das versões propostas e dos módulos de vocabulário e matriz de co-ocorrências do GloVe, juntamente com o speed-up obtido.*

este experimento foi utilizado como *corpus* um extrato da Wikipedia (em inglês) de 2014 com aproximadamente 8.5 GiB, com um vocabulário de 5.021.032 termos e um total de 1.546.379.363 *tokens*. A Tabela 4.11 apresenta os valores de acurácia medidos em um dos testes realizados com os *embeddings* produzidos. Nesse teste utilizou-se o *Google Analogy Test Set* e os resultados foram comparados com os resultados obtidos através do mesmo conjunto de testes, usando os *embeddings* gerados pelo *GloVe* e pelo *Word2Vec*, treinados no mesmo *corpus* e vocabulário.

	RI 300 P1	RI 1024 P1	GloVe 300 P14	Word2Vec 300 P5
Capitais de Países Comuns	**27,87%** (141/506)	**27,87%** (141/506)	0,00% (0/506)	0,00% (0/506)
Capitais de Todos os Países	16,38% (741/4524)	**17,88%** (809/4524)	0,00% (0/2039)	0,05% (0/2039)
Moedas	0,69% (6/866)	0,92% (8/866)	0,25% (8/808)	**5,94%** (48/808)
Cidades em Estados	23,15% (571/2467)	**26,02%** (642/2467)	0,00% (0/1798)	0,00% (0/1798)
Família	56,13% (284/506)	55,14% (279/506)	10,08% (51/506)	**87,35%** (442/506)
Adjetivos × Advérbios	3,12% (31/992)	3,02% (30/992)	0,81% (8/992)	**27,02%** (268/992)
Antônimos	6,16% (50/812)	1,48% (12/812)	1,48% (12/812)	**32,51%** (264/812)
Advérbios	14,11% (188/1332)	14,49% (193/1332)	0,98% (13/1332)	**89,34%** (1190/1332)
Adjetivos (Grau Superlativo)	4,10% (46/1122)	4,19% (47/1122)	0,36% (4/1122)	**65,33%** (733/1122)
Particípio	3,03% (32/1056)	3,88% (41/1056)	1,70% (18/1056)	**73,77%** (779/1056)
Gentílicos	17,76% (284/1599)	**19,26%** (308/1599)	0,00% (0/1299)	0,23% (3/1299)
Pretérito	10,77% (168/1560)	10,77% (168/1560)	2,37% (37/1560)	**65,64%** (1024/1560)
Plural	6,68% (89/1332)	7,06% (94/1332)	4,58% (61/1332)	**89,79%** (1196/1332)
Plural de Verbos	4,37% (38/870)	5,29% (46/870)	4,02% (35/870)	**72,76%** (633/870)
Questões vistas/total	**100,00%** (19544/19544)	**100,00%** (19544/19544)	82,03% (16032/19544)	82,03% (16032/19544)
Acurácia Semântica	19,65% (1743/8869)	**21,19%** (1879/8869)	0,94% (53/5657)	8,68% (491/5657)
Acurácia Sintática	8,67% (926/10675)	9,26% (989/10675)	1,81% (188/10375)	**58,70%** (6090/10375)
Acurácia Total	13,66% (2669/19544)	14,67% (2868/19544)	1,50% (241/16032)	**41,05%** (6581/16032)

Tabela 4.11: *Acurácia dos vetores produzidos com os algoritmos Random Indexing, GloVe e Word2Vec*

As colunas representam, em ordem da esquerda para a direita, os valores obtidos com os vetores gerados pelo algoritmo *Random Indexing* com 300 e 1024 dimensões, respectivamente, produzidos em uma única época (P1); os vetores gerados pelo *GloVe* com 300 dimensões e treinados em 14 épocas, sendo esses vetores obtidos na época com menor custo de um total de 25 épocas; e os vetores gerados pelo *Word2Vec* com 300 dimensões e treinados em 5 épocas.

O teste realizado consiste numa tarefa de analogia baseada no dataset[6] proposto por Mikolov [45], denominado *Google Analogy Test Set*, que objetiva avaliar a analogia como um grau de similaridade existente nos *embeddings*. Desta forma, para cada três palavras existentes no *dataset*, uma quarta é solicitada ao modelo a partir do vocabulário disponível. Com os resultados, pode-se observar uma qualidade dos *embeddings*, como a acurácia sintática, relacionada com a capacidade dos vetores representarem uma similaridade estrutural, como no caso de plural; ou a acurácia semântica, onde os vetores conseguem representar associações de sentido, como sinonímia.

Como pode ser observado na Tabela 4.11, o algoritmo *Random Indexing*, utilizando nossa implementação de tabela *hash*, produz os vetores de 100% dos tokens do vocabulário. Diferentemente do que ocorre em [55], uma proposta de implementação na GPU do mesmo algoritmo, onde, devido a limitações do hardware e da abordagem adotada, parte do vocabulário é descartada. Produzir 100% dos tokens do vocabulário permite, por exemplo, que tarefas de NLP tenham maior acurácia, ao diminuir a probabilidade de um termo não ser encontrado, tendo assim, que ser substituído por um termo "coringa".

Outro ponto que se destaca é a quantidade de recursos computacionais necessários para a produção dos *embeddings*. Com o *Random Indexing* foi possível gerar os *embeddings* na máquina onde os próprios experimentos foram programados, a máquina denominada como **exp**. Contudo, a quantidade de memória disponível nessa máquina não foi suficiente para conseguir executar o *GloVe* com um conjunto de dados do tamanho do *dataset* utilizado. Foi necessário utilizar a máquina denominada como **vm**, com 88 GB de RAM, para que os dados e o programa pudessem ser executados, consumindo-se mais de 24h para o treinamento completo.

O *Word2Vec* foi executado na máquina denominada **ufg** devido à quantidade de processamento disponível, visto que a versão utilizada do programa foi a disponibilizada pelo Google no repositório oficial do projeto, que suporta apenas CPU.

Vale destacar que o *Google Analogy Test Set* foi desenvolvido por Mikolov et. al [45] durante sua pesquisa sobre o Word2Vec. O intuito era demonstrar a qualidade dos *embeddings* produzidos. Contudo, alguns problemas são destacados na Wiki da ACL Web [1]:

- O teste não é balanceado, contendo entre 20 e 70 pares por categoria, com um número diferente de relacionamentos semânticos e sintáticos;
- A categoria `country:capital` (*Capitais de Todos os Países* na Tabela 4.11) contém mais de 50% de todas as relações semânticas existentes no conjunto de testes;
- Por não ser balanceado, pode privilegiar alguns formatos de *embeddings*;

[6] Google Analogy Test Set http://download.tensorflow.org/data/questions-words.txt

- A acurácia também depende do método utilizado para resolver a analogia.

CAPÍTULO 5

Conclusão

Neste trabalho foram apresentados algoritmos que permitem a construção de tabelas *hash* de forma eficiente, com foco principal nas arquiteturas *manycore*, que devido à organização da própria arquitetura, dificulta o uso de estruturas de dados dessa natureza. Foram apresentados também algoritmos que utilizam esta tabela *hash* para auxiliar e melhorar o desempenho na realização de tarefas de pré-processamento, como a limpeza de termos irrelevantes, *stemming* e a extração do vocabulário; na geração da matriz de coocorrência; e na construção de *embeddings*, representações ricas semanticamente.

Os algoritmos e os experimentos desenvolvidos relacionados aos *embeddings* estabelecem uma abordagem para a representação de termos utilizando-se vetores densos, que pode ser utilizada facilmente pelos principais algoritmos de aprendizado de máquina atuais, com grande ganho de desempenho através do uso de técnicas de paralelização do código e uso da GPU em arquiteturas que suportem CUDA. O modelo de *embeddings* sugerido permite a ampliação da base de representações e a atualização constante dos vetores existentes, evitando a degradação da qualidade dos modelos gerados que fazem uso de *embeddings*, devido à desatualização das representações vetoriais e à complexidade e tempo necessários para a sua total reconstrução.

De acordo com os experimentos realizados e os resultados obtidos, pode-se avaliar que os algoritmos propostos, tanto para a estrutura de dados quanto para os seus casos de uso, são eficientes e eficazes, o que permite que estas soluções sejam utilizadas para auxiliar a construção de outros trabalhos acadêmicos futuros, e até mesmo soluções de terceiros.

Desta forma, após a conclusão desta pesquisa, espera-se contribuir com a disponibilização, para a comunidade científica, da implementação paralela dos algoritmos projetados, que poderá ser utilizada em experimentos e sistemas, beneficiando-os com ganhos em tempo e performance, da mesma forma que se obteve neste trabalho.

5.1 Limitações e Trabalhos Futuros

Devido aos limites de tempo pertinentes a um mestrado e aos recursos computacionais disponíveis, algumas áreas de interesse não foram analisadas, postergando-se seu estudo para um momento futuro e adequado. Dentre estas destaca-se:

- Experimentar outros mecanismos para avaliação da qualidade dos *embeddings*, considerando-se aplicações especializadas, que fazem uso intensivo ou dependem totalmente dessa forma de representação textual;
- Avaliar melhores abordagens para o gerenciamento de memória na GPU. Permitindo-se a troca dinâmica de dados, de modo semelhante ao que ocorre com a *Unified Memory* presente atualmente no CUDA, porém, removendo-se os *overheads* existentes;
- Avaliar os algoritmos projetados e os programas desenvolvidos utilizando-se bases de dados textuais maiores e mais complexas, o que não foi possível devido ao *hardware* disponível para o desenvolvimento e realização de experimentos;
- Analisar as propriedades e os benefícios de sistemas multi-GPUs, experimentando-se com os dispositivos, barramentos, arquiteturas e algoritmos existentes.

Com a continuidade dos estudos e o aprimoramento das técnicas e algoritmos já desenvolvidos, espera-se alcançar melhorias de desempenho, auxiliando-se outros pesquisadores e desenvolvedores nas soluções de problemas em áreas como a mineração de textos e o processamento de linguagem natural.

Bibliografia

[1] ACL WEB. **Google analogy test set (State of the art)**, (acessado em 13/08/2020). https://aclweb.org/aclwiki/Google_analogy_test_set_(State_of_the_art).

[2] AMDAHL, GENE MYRON. **Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities**. In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), p. 483?485, New York, NY, USA, 1967. Association for Computing Machinery.

[3] ANDRADE, G.; VIEGAS, F.; RAMOS, G. S.; ALMEIDA, J.; ROCHA, L.; GONÇALVES, M.; FERREIRA, R. **GPU-NB: A Fast CUDA-Based Implementation of Naive Bayes**. In: *2013 25th International Symposium on Computer Architecture and High Performance Computing*, p. 168–175, 2013.

[4] ANDROUTSOPOULOS, I.; KOUTSIAS, J.; CHANDRINOS, K. V.; PALIOURAS, G.; SPYROPOULOS, C. D. **An evaluation of Naive Bayesian anti-spam filtering**. *arXiv:cs/0006013*, Jun 2000. arXiv: cs/0006013.

[5] APPLEBY, A. **MurMurHash's GitHub page**, (acessado em 04/05/2019). https://github.com/aappleby/smhasher.

[6] BALAKRISHNAN, V.; LLOYD-YEMOH, E. **Stemming and lemmatization: a comparison of retrieval performances**. 2014.

[7] BARROS, C. C.; MARTINS, W. **Acelerando a construção de vocabulário e matriz de co-ocorrência em bases textuais**. In: *Anais Principais do XX Simpósio em Sistemas Computacionais de Alto Desempenho*, p. 418–429. SBC, 2019.

[8] BENGIO, Y.; DUCHARME, R.; VINCENT, P.; JAUVIN, C. **A neural probabilistic language model**. *Journal of machine learning research*, 3(Feb):1137–1155, 2003.

[9] BENGIO, Y.; LAMBLIN, P.; POPOVICI, D.; LAROCHELLE, H. **Greedy layer-wise training of deep networks**. In: *Proceedings of the 19th International Conference on Neural Information Processing Systems*, NIPS'06, p. 153–160, Cambridge, MA, USA, 2006. MIT Press.

[10] BOJANOWSKI, P.; GRAVE, E.; JOULIN, A.; MIKOLOV, T. **Enriching Word Vectors with Subword Information**. *arXiv:1607.04606 [cs]*, Jul 2016. arXiv: 1607.04606.

[11] CHELBA, C.; MIKOLOV, T.; SCHUSTER, M.; GE, Q.; BRANTS, T.; KOEHN, P.; ROBINSON, T. **One billion word benchmark for measuring progress in statistical language modeling**. *arXiv preprint arXiv:1312.3005*, 2013.

[12] CHENG, J.; GROSSMAN, M.; MCKERCHER, T. **Professional CUDA C Programming**. Wrox, 1 edition, 09 2014.

[13] COLLOBERT, R.; WESTON, J. **A unified architecture for natural language processing: Deep neural networks with multitask learning**. In: *Proceedings of the 25th International Conference on Machine Learning*, ICML '08, p. 160–167, New York, NY, USA, 2008. ACM.

[14] COLLOBERT, R.; WESTON, J.; BOTTOU, L.; KARLEN, M.; KAVUKCUOGLU, K.; KUKSA, P. P. **Natural language processing (almost) from scratch**. *CoRR*, abs/1103.0398, 2011.

[15] DA SILVA, I. N.; SPATTI, D. H.; FLAUZINO, R. A. **Redes Neurais Artificiais para Engenharia e Ciências Aplicadas**. ARTLIBER, 2010.

[16] DEERWESTER, S.; DUMAIS, S. T.; FURNAS, G. W.; LANDAUER, T. K.; HARSHMAN, R. **Indexing by latent semantic analysis**. *Journal of the American Society for Information Science*, 41(6):391–407.

[17] DOURADO, I.; GALANTE, R.; ANDRÉ GONÇALVES, M.; TORRES, R. **Bag of textual graphs (botg): A general graph-based text representation model**. *Journal of the Association for Information Science and Technology*, 01 2019.

[18] ENBODY, R. J.; DU, H. C. **Dynamic Hashing Schemes**. *ACM Comput. Surv.*, 20(2):850?113, July 1988.

[19] FAYYAD, U.; PIATETSKY-SHAPIRO, G.; SMYTH, P. **From data mining to knowledge discovery in databases**. *AI Magazine*, 17(3):37?37, Mar 1996.

[20] FOWLER, G.; NOLL, L. C.; VO, K.-P.; EASTLAKE, D.; HANSEN, T. **The FNV Non-Cryptographic Hash Algorithm**. *IETF Informational Internet Draft*.

[21] GLASKOWSKY, P. N. **NVIDIA's Fermi: The First Complete GPU Computing Architecture**. Technical report, NVIDIA Corporation, 2009.

[22] GOLDBERG, Y.; HIRST, G. **Neural Network Methods in Natural Language Processing (Synthesis Lectures on Human Language Technologies)**. Morgan & Claypool Publishers, 1 edition, 2017.

[23] GOODFELLOW, I.; BENGIO, Y.; COURVILLE, A. **Deep Learning**. MIT Press, 2016. http://www.deeplearningbook.org.

[24] GRAVE, E.; JOULIN, A.; CISSÉ, M.; GRANGIER, D.; JÉGOU, H. **Efficient softmax approximation for GPUs**. *arXiv preprint arXiv:1609.04309*, 2016.

[25] GROUP, S. M. **Babbage's analytical engine, 1834-1871. (trial model): Science museum group collection**, (acessado em 10/10/2020).

[26] GUSTAFSON, JOHN L.. **Reevaluating Amdahl's Law**. *Commun. ACM*, 31(5):532?533, May 1988.

[27] GUTHRIE, D.; ALLISON, B.; LIU, W.; GUTHRIE, L.; WILKS, Y. **A Closer Look at Skip-gram Modelling**. 01 2006.

[28] HAYKIN, S. **Neural Networks: A Comprehensive Foundation**. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition, 1998.

[29] IAN BOYD. **StackOverflow's Answer — Which hashing algorithm is best for uniqueness and speed?**, (acessado em 19/09/2020). https://softwareengineering.stackexchange.com/a/145633.

[30] INGASON, A.; HELGADÓTTIR, S.; LOFTSSON, H.; RÖGNVALDSSON, E. **A mixed method lemmatization algorithm using a hierarchy of linguistic identities (holi)**. volume 5221, p. 205–216, 01 2008.

[31] INTEL CORPORATION. **Intel Xeon Phi Coprocessor Architecture Overview**. https://software.intel.com/sites/default/files/Intel%C2%AE_Xeon_Phi%E2%84%A2_Coprocessor_Architecture_Overview.pdf. Acessado em: 2017-10-16.

[32] INTEL CORPORATION. **Intel Xeon Phi Product Family**. https://www.intel.com/content/www/us/en/products/processors/xeon-phi.html. Acessado em: 2017-10-16.

[33] INTEL CORPORATION. **Intel Xeon Phi Specification**. https://ark.intel.com/products/95830/Intel-Xeon-Phi-Processor-7290-16GB-1_50-GHz-72-core. Acessado em: 2017-10-16.

[34] JASON RENNIE. **20 Newsgroups**, (acessado em 05/11/2020). http://qwone.com/~jason/20Newsgroups/.

[35] JIVANI, A. G.; OTHERS. **A Comparative Study of Stemming Algorithms**. *Int. J. Comp. Tech. Appl*, 2(6):1930–1938, 2011.

[36] JOULIN, A.; GRAVE, E.; BOJANOWSKI, P.; MIKOLOV, T. **Bag of tricks for efficient text classification**. *arXiv:1607.01759 [cs]*, Jul 2016. arXiv: 1607.01759.

[37] JURAFSKY, D.; MARTIN, J. H. **Speech and Language Processing**. Prentice Hall, 2 edition, 2008.

[38] KANERVA, P.; KRISTOFERSON, J.; HOLST, A. **Random Indexing of Text Samples for Latent Semantic Analysis**. In: *Proceedings of the Annual Meeting of the Cognitive Science Society*, 2000.

[39] KERNIGHAN, B. W.; RITCHIE, D. M. **The C Programming Language**. Prentice-Hall, Inc., USA, 1978.

[40] KIRK, D. B.; MEI W. HWU, W. **Programming Massively Parallel Processors: A Hands-on Approach**. Morgan Kaufmann, 3 edition, 12 2016.

[41] KURGAN, L. A.; MUSILEK, P. **A survey of Knowledge Discovery and Data Mining process models**. *The Knowledge Engineering Review*, 21(1):1?24, 2006.

[42] LANDON CURT NOLL. **FNV Hash**, (acessado em 19/09/2020). http://www.isthe.com/chongo/tech/comp/fnv/index.html.

[43] LANG, K. **Newsweeder: Learning to filter netnews**. In: *Proceedings of the Twelfth International Conference on Machine Learning*, p. 331–339, 1995.

[44] LEVY, O.; GOLDBERG, Y.; DAGAN, I. **Improving distributional similarity with lessons learned from word embeddings**. *Transactions of the Association for Computational Linguistics*, 3:211–225, 2015.

[45] MIKOLOV, T.; CHEN, K.; CORRADO, G.; DEAN, J. **Efficient Estimation of Word Representations in Vector Space**. 2013, 01 2013.

[46] MIKOLOV, T.; SUTSKEVER, I.; CHEN, K.; CORRADO, G. S.; DEAN, J. **Distributed representations of words and phrases and their compositionality**. In: *Advances in neural information processing systems*, p. 3111–3119, 2013.

[47] MORIN, F.; BENGIO, Y. **Hierarchical probabilistic neural network language model.** In: *Aistats*, volume 5, p. 246–252. Citeseer, 2005.

[48] NEEDHAM, M. **Python/scikit-learn: Calculating TF/IDF on How I met your mother transcripts**. http://www.markhneedham.com/blog/2015/02/15/pythonscikit-learn-calculating-tfidf-on-how-i-met-your-mother-transcripts/. Acessado em : 2017-10-16.

[49] NVIDIA. **GeForce 256 - The World's First GPU**, (acessado em 11/10/2020). https://web.archive.org/web/20040414145655/http://www.nvidia.com/page/geforce256.html.

[50] OWENS, J. D.; LUEBKE, D.; GOVINDARAJU, N.; HARRIS, M.; KRÜGER, J.; LEFOHN, A. E.; PURCELL, T. J. **A survey of general-purpose computation on graphics hardware**. In: *Computer graphics forum*, volume 26, p. 80–113. Wiley Online Library, 2007.

[51] PAIK, J. H.; PAL, D.; PARUI, S. K. **A novel corpus-based stemming algorithm using co-occurrence statistics**. In: *Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR ?11, p. 863?872. ACM, 2011. event-place: Beijing, China.

[52] PANG, B.; LEE, L. **Opinion Mining and Sentiment Analysis**. *Foundations and Trends® in Information Retrieval*, 2(1?2):1–135, 2008.

[53] PENNINGTON, J.; SOCHER, R.; MANNING, C. **Glove: Global vectors for word representation**. In: *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, p. 1532–1543, 2014.

[54] PLISSON, J.; LAVRAC, N.; MLADENIC, D.; OTHERS. **A rule based approach to word lemmatization**. In: *Proceedings of IS*, volume 3, p. 83–86, 2004.

[55] POLOK, L.; SMRZ, P. **Implementing Random Indexing on GPU**. In: *Proceedings of the 19th High Performance Computing Symposia*, p. 134–142, 2011.

[56] PORTER, M. F. **An algorithm for suffix stripping**. *Program*, 14(3):130–137, 1980.

[57] PORTER, M. F. **Snowball: A language for stemming algorithms**, 2011. http://snowball.tartarus.org/texts/introduction.html.

[58] RONG, X. **word2vec Parameter Learning Explained**. 01 2016.

[59] SAHAMI, M.; DUMAIS, S.; HECKERMAN, D.; HORVITZ, E. **A Bayesian approach to filtering junk e-mail**. In: *Learning for Text Categorization: Papers from the 1998 workshop*, volume 62, p. 98–105. Madison, Wisconsin, 1998.

[60] SAHLGREN, M. **An Introduction to Random Indexing**. *Methods and applications of semantic indexing workshop at the 7th international conference on terminology and knowledge engineering*, 2005.

[61] SANDERS, J.; KANDROT, E. **CUDA by Example: An Introduction to General-Purpose GPU Programming**. Addison-Wesley Professional, 1 edition, 07 2010.

[62] 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.

[63] SINGH, B.; YADAV, I.; AGARWAL, S.; PRASAD, R. **An efficient word searching algorithm through splitting and hashing the offline text**. In: *2009 International Conference on Advances in Recent Technologies in Communication and Computing*, p. 387–389, 2009.

[64] SINGH, J.; GUPTA, V. **Text Stemming: Approaches, Applications, and Challenges**. *ACM Computing Surveys*, 49(3):45:1–45:46, 09 2016.

[65] SIPSER, M. **Introdução à Teoria da Computação**. Cengage Learning, 2005.

[66] SOARES, M. V. B.; PRATI, R. C.; MONARD, M. C. **PreText II: Descrição da Reestruturação da Ferramenta de Pré-Processamento de Textos**. Technical report, Universidade de São Paulo – Instituto de Ciências Matemáticas e de Computação - ICMC, 2008.

[67] STALLINGS, W. **Computer Organization and Architecture: Designing for Performance**. Prentice Hall Press, USA, 8th edition, 2009.

[68] VIEGAS, F.; GONÇALVES, M. A.; MARTINS, W.; ROCHA, L. **Parallel Lazy Semi-Naive Bayes Strategies for Effective and Efficient Document Classification**. In: *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*, p. 1071–1080. ACM.

[69] YIGIT, O. **Hash functions**, (acessado em 19/09/2020). https://web.archive.org/web/20200919052448/http://www.cse.yorku.ca/~oz/hash.html.

[70] ZHANG, Y.; JIN, R.; ZHOU, Z.-H. **Understanding bag-of-words model: a statistical framework**. *International Journal of Machine Learning and Cybernetics*, 1(1):43?52, Dec 2010.

[71] ZHANG, Y.; MUELLER, F.; CUI, X.; POTOK, T. **GPU-Accelerated Text Mining**. 01 2009.

[72] ZIPF, G. **Human behaviour and the principle of least-effort**. 1949.