Acelerando a construção de vocabulário e matriz de co-ocorrência em bases textuais

Chayner Cordeiro Barros, Wellington Santos Martins

[1]Instituto de Informática – Universidade Federal de Goiás (UFG)
Alameda Palmeiras, Quadra D, Campus Samambaia – 74.690-900 – Goiânia – GO – Brazil

{chaynercordeiro,wellington}@inf.ufg.br

Abstract. *Two important preprocessing tasks in natural language processing are vocabulary building and word co-occurrence matrix computation. As datasets get large and non static corpora becomes common, these tasks become increasingly computational demanding. In this article, we present parallel algorithms to extract vocabulary and compute the co-occurrence matrix. These algorithms are mapped to both a multicore (CPU) and a manycore (GPU) architecture. Our experiments using a standard dataset show speedups of up to 21x when compared to a sequential state-of-the-art (GloVe) implementation performing the same tasks.*

Resumo. *Duas tarefas que se destacam no pré-processamento de textos são a construção de um vocabulário e a geração de uma matriz de co-ocorrências de palavras. Para um volume de dados crescente e não estático, estas tarefas requerem um alto custo computacional. Neste artigo, exploramos paralelismo para viabilizar este processamento. Apresentamos algoritmos paralelos para extrair o vocabulário e produzir a matriz de co-ocorrências e implementamos os mesmos em arquiteturas* multicore *e* manycore *(GPU). Os experimentos, utilizando uma base de dados padrão, mostram que nossa implementação consegue ser até 21x mais rápida que uma solução estado-da-arte (GloVe) sequencial na realização das mesmas tarefas.*

1. Introdução

A mineração de texto (*text mining*) se caracteriza pela extração de informações a partir de dados textuais, nos mais diversos formatos, objetivando a classificação desta informação e a produção de conhecimento. 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 [Pang and Lee 2008]. 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 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 esteja estruturado num formato mais fácil de ser manipulado computacionalmente. Estes procedimentos são denominados de "pré-processamento" e constituem uma fase crucial no processo de extração [Manning and Schütze 1999].

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 mineração de textos, 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 [Weiss et al. 2005].

Algumas tarefas que se destacam no pré-processamento de textos são a construção de um vocabulário e a geração de uma matriz de co-ocorrências de termos (palavras). O vocabulário permite identificarmos os termos únicos encontrados no corpus. Já a matriz de co-ocorrências permite que associações entre palavras, num dado contexto, sejam reveladas. Algumas das aplicações que utilizam estas tarefas são extração de palavras-chave e representações semânticas, avaliação automática de resumos e traduções, entre outras [Pennington et al. 2014, Schuetze 1997, Doddington 2002]

Mesmo que a obtenção do vocabulário e da matriz de co-ocorrências seja feita em uma única leitura de todos os arquivos da coleção de documentos, essa leitura em grandes coleções de documentos pode ser computacionalmente dispendiosa. Além disso, para muitas aplicações de processamento de linguagem natural (NLP), estes dados não são estáticos. Por exemplo, nos mecanismos de busca ou redes sociais as informações estão em constante mudança, o que requer uma atualização frequente, e rápida, destas estruturas de dados. Esta necessidade de pré-processamento constante, e envolvendo uma quantidade crescente de dados, pode se valer do poder computacional das arquiteturas modernas, altamente paralelas, para viabilizar este processamento.

Neste artigo, apresentamos três algoritmos e implementações para extrair o vocabulário e produzir a matriz de co-ocorrências. O primeiro algoritmo, sequencial, serviu de base de comparação para os outros dois, paralelos. Todos fazem uso de uma estrutura de dados do tipo *Hash* que armazena o vocabulário e a matriz de co-ocorrências. O primeiro algoritmo paralelo é uma extensão do sequencial, fazendo uso dos vários núcleos de uma CPU *multicore*. O segundo algoritmo paralelo explora paralelismo de granularidade fina e é mapeado para uma arquitetura *manycore* do tipo GPU. Um aspecto importante das propostas é o tratamento dado no uso distribuído da estrutura de *Hash*. Para avaliar a qualidade dos algoritmos paralelos, comparamos as implementações dos mesmos com o equivalente sequencial proposto, e com uma implementação sequencial estado-da-arte (GloVe – *Global Vectors for Word Representation*) [Pennington et al. 2014]) que também gera o vocabulário e a matriz de co-ocorrências.

2. Trabalhos Relacionados

A matriz de co-ocorrências de palavra-palavra, comumente denominada *word-word co-occurrence matrix*, é um artefato muito utilizado em diversos modelos de representação, tarefas de NLP (*Natural Language Processing* – processamento de linguagem natural), tradução por máquina, mineração de texto etc. A construção desta matriz é uma atividade dispendiosa computacionalmente, porque exige a análise da relação entre os termos existentes no corpus, podendo consumir muita memória caso o vocabulário e a janela de contexto empregada sejam muito grandes.

O modelo de representação vetorial GloVe [Pennington et al. 2014] fatora a ma-

triz de co-ocorrências para construir os vetores que representam os termos enumerados no vocabulário, e obtém dela a relação semântica entre esses termos. Pennington acelerou o cálculo da matriz de co-ocorrências palavra-palavra com uma solução que busca encontrar regiões de alta densidade na matriz, para então calculá-las de forma separada. Uma outra abordagem [Lin 2009] propõe o uso de MapReduce para o cálculo dessas matrizes. Porém, nenhuma das duas soluções anteriores aproveitam o poder computacional que as GPU's oferecem. [Khuc et al. 2012] também aplicam matrizes de co-ocorrências na tarefa de análise de sentimentos. Eles utilizam a matriz com bi-gramas como ferramenta para auxiliar seu algoritmo na detecção da polaridade de *tweets*.

A nossa solução é baseada numa estrutura de dados do tipo Hash que é utilizada para armazenar tanto o vocabulário quanto a matriz de co-ocorrências. Além de ser bastante eficiente para o armazenamento e recuperação das informações, esta solução se adapta facilmente a diferentes arquiteturas, como foi o caso das implementações *multicore* (CPU) e *manycore* (GPU) propostas. Além disso, as soluções propostas se adaptam à diferentes recursos computacionais, funcionando assim, por exemplo, com uma quantidade menor de memória disponível, embora a maior disponibilidade desse recurso aumente a eficiência dos algoritmos propostos.

3. Arquitetura da GPU

Em seguida, apresentamos uma breve descrição da arquitetura de uma GPU e seu modelo de programação correspondente (Ver [Kirk and Wen-mei 2012] para mais detalhes). A Figura 1 mostra uma visão de alto nível da arquitetura de uma GPU. A arquitetura da GPU possui dois níveis de paralelismo, nos quais os multi-processadores (SMs) de fluxo estão no primeiro nível e os processadores de fluxo (SPs) ficam dentro de cada multi-processador. Assim, um programa paralelo deve ser primeiro dividido em blocos de computação para serem executados independentemente nos SMs, sem se comunicarem entre si. Esses blocos são divididos em tarefas menores (*threads*) executadas nos SPs, mas com cada *thread* podendo se comunicar com outras *threads* no mesmo bloco. Cada uma dessas *threads* tem acesso a uma memória global maior, bem como a uma memória compartilhada pequena, mas rápida, e os registradores.

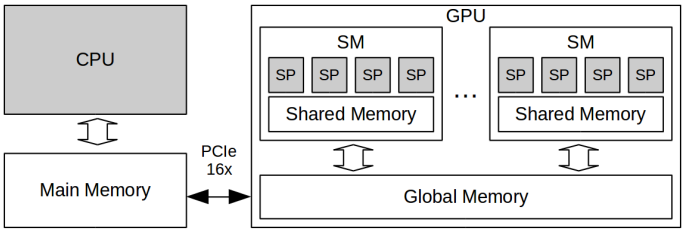

Figura 1. Visão geral da arquitetura de uma GPU

A GPU suporta milhares de *threads* concorrentes e, ao contrário das *threads* da CPU, a sobrecarga de criação e comutação destas *threads* é insignificante. Para ocultar a alta latência da memória global, é importante ter mais *threads* do que número de SPs e ter *threads* que acessem endereços de memória consecutivos que possam ser facilmente reunidos. Outro canal importante de movimentação de dados é a conexão PCIExpress, na qual CPU e GPU podem trocar dados entre o espaço de endereços de cada um, mas em uma velocidade muito mais lenta. O modelo de programação da GPU requer que parte da

aplicação seja executada na CPU, enquanto a parte de computação intensiva é acelerada pela GPU. Um programa de GPU expõe o paralelismo através de uma função especial chamada de função kernel. Durante a implementação, o programador pode configurar o número de *threads* a serem utilizados. As *threads* executam cálculos paralelos do kernel e são organizadas em grupos (blocos de *threads*) que compõem uma estrutura de grade. Quando um kernel é iniciado, os blocos dentro de uma grade são distribuídos em SMs ociosos enquanto as *threads* são mapeados para os SPs.

4. Estrutura de Dados e Algoritmos Propostos

São apresentados três algoritmos para a construção do vocabulário e geração da matriz de co-ocorrências. Entretanto, todos estes algoritmos fazem uso de uma estrutura de dados do tipo Hash que é utilizada para armazenar tanto o vocabulário quanto a matriz de co-ocorrências. Os algoritmos extraem o vocabulário V e a matriz de co-ocorrências M de uma coleção de segmentos de texto. O termo "segmento" é uma definição genérica, que varia de acordo com a aplicação, sendo a implementação livre para definir os limites e tamanho de cada segmento. Nas seções seguintes apresentamos a estrutura de dados e os três algoritmos propostos.

4.1. Estrutura de Dados - Hash

Para extrairmos o vocabulário e a matriz de co-ocorrência de forma eficiente, precisamos utilizar estruturas de dados capazes de representar esses elementos e que permitam um acesso rápido aos elementos armazenados nelas. Uma forma comum de representar o vocabulário é criar uma tabela *hash*, associando uma `id` com o termo e a sua respectiva frequência. A mesma representação pode ser utilizada para matrizes esparsas, como é o caso da matriz de co-ocorrência.

Contudo, um fator limitante é a dificuldade de se construir uma implementação eficiente de tabelas *hash* utilizando a GPU, como pode ser observado em [Alcantara et al. 2009], que demonstra o algoritmo Cuckoo. Desta forma, optou-se por produzir uma implementação especializada, que utiliza operações atômicas para determinar e controlar as colisões dentro da tabela, assim como controlar o acesso concorrente entre as *threads* da GPU. O Algoritmo 1 demonstra como a inserção é feita na tabela hash que implementa o vocabulário. Ela utiliza endereçamento aberto como forma de tratamento das colisões entre os elementos.

As inserções são realizadas buscando consecutivamente por uma posição livre na tabela. A concorrência é administrada pelo uso das operações atômicas: `atomicCAS` (*Atomic Compare and Swap*), que compara o valor no endereço fornecido no primeiro parâmetro com o segundo parâmetro fornecido, e em caso de igualdade, o substitui pelo valor no terceiro parâmetro, retornando o valor antigo existente no endereço; `atomicAdd`, que incrementa o valor contido no endereço fornecido como primeiro parâmetro com o valor fornecido no segundo parâmetro.

O algoritmo testa se a posição calculada (linha 3) já possui o elemento que queremos inserir. Em caso positivo, apenas incrementa o contador existente (linha 5) e encerra. Em caso negativo, devemos testar novamente a posição, pois, na nossa implementação, se a posição estiver livre ela estará marcada com '0'. Desta forma, analisamos todas as posições subsequentes, desde a atual, até que uma posição esteja livre ou contenha

o elemento desejado. Como pode ser observado no algoritmo, em caso de colisão entre elementos, a tabela é tratada como uma lista circular. A cópia do *token* (no caso do vocabulário) é feita apenas quando a posição encontrada for vazia (linhas 14 e 17).

```
   Entrada: tabela hash H
   inteiro id representando a identificação ou hashcode do elemento
   elemento q que será inserido na tabela
 1 h ← hash(id)
 2 pos ← h  mod |H|
 3 v ← atomicCAS (H(pos), h, h)
 4 se v = h então
 5 |   atomicAdd (H(pos).count, 1)
 6 |   retorna ok
 7 senão
 8 |   para i ← pos; i < |H| faça
 9 |   |   v ← atomicCAS (H(i), h, h)
10 |   |   se v = h então
11 |   |   |   atomicAdd (H(i).count, 1)
12 |   |   |   retorna ok
13 |   |   senão
14 |   |   |   v ← atomicCAS (H(i), 0, h)
15 |   |   |   se v = 0 então
16 |   |   |   |   atomicAdd (H(i).count, 1)
17 |   |   |   |   copia o elemento q para dentro da estrutura
18 |   |   |   |   retorna ok
19 |   |   |   senão
20 |   |   |   |   v ← atomicCAS (H(i), h, h)
21 |   |   |   |   se v = h então
22 |   |   |   |   |   atomicAdd (H(i).count, 1)
23 |   |   |   |   |   retorna ok
24 |   para i ← 0; i < pos faça
25 |   |   repita os mesmos passos do loop anterior
26 |   retorna tabela cheia
```

Algoritmo 1: Algoritmo de inserção na tabela *hash*

4.2. Algoritmo Sequencial - Base

O algoritmo sequencial serve de base de comparação para avaliarmos o desempenho dos algoritmos paralelos, Todos algoritmos fazem uso da Hash para armazenar o vocabulário e matriz de co-ocorrência. O vocabulário armazena um *token* que 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 e um contador de vezes que esse termo foi encontrado em todos os documentos da coleção. A matriz

de co-ocorrência opera de forma semelhante, com a diferença de que são armazenadas as quantidades de vezes em que pares de termos co-ocorreram dentro de uma janela pré-definida. Observe na Figura 2 o documento com o seguinte texto: "*The quick brown fox jumps over the lazy dog*". 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.

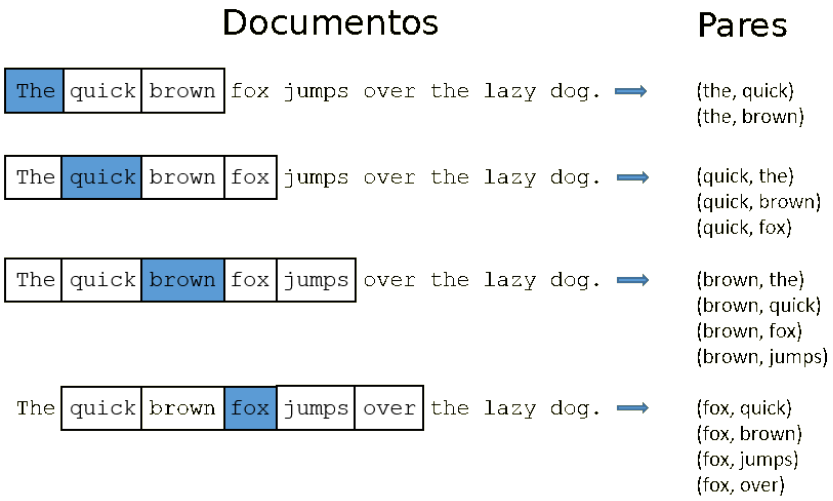

Figura 2. Exemplo de janela de contexto simétrica de tamanho 2. [McCormick 2017]

A solução sequencial é mostrada no Algoritmo 2. O algoritmo considera 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. O algoritmo começa com a inicialização das duas estruturas essenciais para o problema, o vocabulário V e a matriz de co-ocorrências M. Então, uma iteração é realizada para processar cada arquivo de texto a, que contém os documentos. 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. O algoritmo funciona tanto em janelas simétricas quanto assimétricas, isto é, aquelas onde apenas uma das direções é considerada.

4.3. Algoritmo Paralelo - *Multicore*

A principal melhoria no algoritmo *multicore*, Algoritmo 3, é a distribuição dos arquivos de dados entre as unidades de processamento (núcleos) disponíveis (CPU no algoritmo). Os arquivos são carregados para a memória e encaminhados para uma das unidades de processamento t. As estruturas de dados que armazenam o vocabulário V e a matriz de co-ocorrências 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 desempenho. Ao término do processamento paralelo, o vocabulário V e a matriz de co-ocorrências M 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.

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

```
1  V ← [∞] : token
2  M ← [∞] : tupla
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
```

Algoritmo 2: Algoritmo Sequencial

4.4. Algoritmo Paralelo - Manycore

O algoritmo que explora granularidade fina (*manycore* - GPU), Algoritmo 4, 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-ocorrências (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 segmentos possuem tamanho fixo, diferente do que ocorre nas versões sequencial e *multicore*, descritas anteriormente. Os segmentos são processados em lotes de B_{size} segmentos, 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 segmentos 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-ocorrências 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 segmentos 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.

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-ocorrências
booleano j_{sim} indicando se a janela é simétrica
Saída: vocabulário $\rightarrow V_T$, matriz de co-ocorrências $\rightarrow M_T$

```
1  V ← [CPU, ∞] : token
2  M ← [CPU, ∞] : tupla
3  para cada a ∈ A faça
4      para cada t ∈ CPU faça
5          leia a_t para a memória
6          para cada d ∈ a_t faça em paralelo
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                 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_{t,w,u}
18 para cada v ∈ V faça
19     V_T ← v
20 para cada m ∈ M faça
21     M_T ← m
```

Algoritmo 3: Algoritmo *Multicore*

Entrada: lista $A[k]$ com os nomes dos k arquivos do corpus
inteiro j_n com o tamanho da janela de co-ocorrências
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-ocorrências $\rightarrow M$

```
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}
```

Algoritmo 4: Algoritmo *Manycore*

5. Experimentos

Para a realização dos experimentos foi utilizado um computador com a seguinte configuração: uma GPU NVIDIA GTX 1050 Ti, processador AMD A10-7850K, 4 cores @ 3.7 GHz, 16 GB de memória DDR3, e um SSD NVMe Samsung 970 EVO como dispositivo de armazenamento. A base de dados utilizada foi a *1 Billion Words Benchmark* [Chelba et al. 2013]. Este conjunto de dados consiste em pouco menos de 1B tokens (cerca de 719M) e é comumente usado para avaliar modelos de linguagem.

Foram construídas três implementações equivalentes, uma para cada algoritmo e arquitetura específica. As implementações paralelas foram comparadas com o algoritmo sequencial proposto e com os métodos otimizados de extração de vocabulário e criação da matriz de co-ocorrências implementados no GloVe [Pennington et al. 2014], executando sequencialmente. Utilizou-se janelas de contexto simétricas de tamanho 2 em todas as medições, inclusive as realizadas com o GloVe em nossos experimentos. Nosso algoritmo não apresenta limitações quanto ao tamanho da janela, mas para efeitos de comparação adotou-se a janela de tamanho 2 para que fosse possível efetuar os experimentos no hardware disponível, sem que fosse necessário diminuir o tamanho do vocabulário.

Todas as medições levam em consideração o tempo gasto com a leitura dos arquivos de dados para a memória, e o processamento desses dados. O tempo gasto com I/O para gravar os resultados foi medido e apresentado separadamente. Para demonstrar o potencial das soluções propostas, utilizou-se um vocabulário com todas as strings, separadas por espaços em branco, encontradas no corpus *1 Billion Words Benchmark*, totalizando 2.438.611 termos únicos. O fato de que a implementação sequencial do GloVe faz operações de I/O simultaneamente com o processamento dos dados, dificulta a separação dos tempos gastos em cada uma dessas atividades, portanto, os resultados são apresentados com os totais obtidos. Para a mensuração do tempo de execução, nenhum dos componentes de performance, seja do sistema operacional, como a cache de disco, ou do hardware, como a cache do processador, foram desabilitados. Utilizou-se a ferramenta de linha de comando do Linux `time` para medir os tempos de execução dos programas de extração de vocabulário e geração da matriz de co-ocorrências do GloVe. Os valores apresentados nas tabelas subsequentes equivalem à média de 10 execuções de cada um dos experimentos e do GloVe realizadas consecutivamente. Nas tabelas encontram-se também os respectivos desvios-padrão. A Tabela 1 apresenta os tempos gastos (em segundos) na execução das versões sequencial, *multicore*, *manycore* e os tempos de execução dos módulos do GloVe que extraem o vocabulário e geram a matriz de co-ocorrências.

Na versão *multicore*, associa-se a cada *thread* um grupo de arquivos de dados. As *threads* são alocadas de acordo com a quantidade de *cores*/*hyperthreads* disponíveis. Como cada *thread* mantém suas tabelas para o vocabulário e para a matriz de co-ocorrências, não há concorrência entre as *threads* por estes recursos, dispensando a necessidade de utilização de *mutexes* ou outros mecanismos de *locking*. A Tabela 1 apresenta os tempos de execução (em segundos) medidos e o *speed-up* obtido em relação à versão sequencial e ao GloVe.

Para a implementação da versão *manycore* foram necessários alguns ajustes, como a definição de limites para os tamanhos dos segmentos de texto analisados. Assim cada *thread* da GPU analisa um bloco de até 1024 bytes de comprimento, finalizado com \n (*line feed*). O tamanho de 1024 bytes do segmento foi escolhido para facilitar o alinha-

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

mento de memória e o cálculo de blocos/*threads* alocados pela GPU. Utilizou-se também CUDA *Streams* para permitir maior escalabilidade em relação às bases de textos, e aumentar o percentual de uso da GPU. Desta forma, os arquivos de dados são divididos em grupos que serão processados independentemente dentro desses *streams*. Nos experimentos realizados com o hardware descrito no início desta seção, cada fluxo analisa um total de 768 MB de texto. Esses e outros valores são facilmente parametrizáveis. Para esses parâmetros, e com a configuração de hardware disponível, são disparados em cada *stream* 768 blocos, com 1024 *threads* por bloco, e cada *thread* analisando 1024 bytes de texto.

Ao final do processamento o vocabulário e a matriz de co-ocorrências são copiados para a memória da CPU, onde o vocabulário é ordenado pela frequência e armazenado em disco. A matriz de co-ocorrências, por sua vez, é gravada diretamente, em formato binário, nos mesmos moldes utilizados pelo GloVe, mas com a diferença de que a memória é mapeada para um arquivo em disco, diminuindo-se o *overhead* causado pelas cópias das páginas de memória entre a aplicação, o *driver* do CUDA e o sistema operacional. A Tabela 1 apresenta os tempos de execução (em segundos) medidos e o *speed-up* obtido em relação à versão sequencial e ao GloVe.

Para determinar a efetividade dos ganhos de performance alcançados pela versão *manycore*, descrita anteriormente, comparou-se os tempos de execução obtidos com os tempos gastos pelos programas que compõem a ferramenta de geração de embeddings do GloVe, que extraem o vocabulário e computam a matriz de co-ocorrências. A Tabela 1 mostra os tempos médios de execução sequencial total do GloVe (incluindo I/O) e apresenta um comparativo entre os tempos obtidos pela nossa solução e o tempo total do GloVe.

O ganho de desempenho obtido nos experimentos se deve não apenas à grande capacidade de processamento da GPU, mas também à 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 duas unidades de cópia (*copy engines*), que são os mecanismos responsáveis por realizar essas transferências assíncronas enquanto a GPU está ocupada realizando algum tipo de processamento.

Outro ponto importante é que nossa solução não necessitou de espaço em disco para efetuar *swapping* das informações em memória. Enquanto que a implementação sequencial do módulo do GloVe que produz a matriz de co-ocorrências consome espaço em disco para armazenar seus arquivos de *overflow*, nossa solução não consumiu espaço

extra algum para processar a base utilizada nos experimentos.

6. Conclusão e Trabalhos Futuros

Apresentamos três algoritmos e implementações para realizar as operações de extração do vocabulário e geração da matriz de co-ocorrências de palavras em bases de dados textuais. Todos os algoritmos fazem uso de uma estrutura de dados do tipo *Hash* que armazena o vocabulário e a matriz de co-ocorrências de modo a facilitar seu acesso durante a execução. Essa *Hash* foi adaptada para cada algoritmo uma vez que o controle de acesso aos dados é dependente da arquitetura-alvo sendo utilizada. As versões paralelas propostas obtiveram ganhos de 3x e 11x em relação a versão sequencial proposta. Além disso, a versão paralela usando o acelerador (*manycore* - GPU) alcançou um *speedup* de 13,35x em relação a implementação sequencial estado-da-arte (GloVe [Pennington et al. 2014]). Estes resultados são encorajadores uma vez que foram utilizadas plataformas de baixo custo e, portanto, bastante acessíveis.

Apesar da base de dados utilizada (*1 Billion Words*) ser padrão neste tipo de experimento, pretendemos investigar o comportamento dos algoritmos propostos em outras bases de dados. A matriz de co-ocorrências não é a melhor medida de similaridade entre duas palavras, pois é baseada na frequência absoluta e, portanto, é muito distorcida. Em vez disso, investigaremos medidas como a Informação Mútua Ponto-a-Ponto (*Pointwise Mutual Information* [Role and Nadif 2011]) que mede a relação entre palavras dentro de um texto comparando a probabilidade de encontrar dois itens juntos com as probabilidades de estarem separadas. Finalmente, pretendemos utilizar estas implementações como parte de uma solução mais abrangente que englobe a geração de *embeddings* (representação vetorial de palavras) para uso na classificação automática de documentos.

Referências

Alcantara, D. A., Sharf, A., Abbasinejad, F., Sengupta, S., Mitzenmacher, M., Owens, J. D., and Amenta, N. (2009). Real-time parallel hashing on the gpu. *ACM Transactions on Graphics (TOG)*, 28(5):154.

Chelba, C., Mikolov, T., Schuster, M., Ge, Q., Brants, T., Koehn, P., and Robinson, T. (2013). One billion word benchmark for measuring progress in statistical language modeling. *arXiv preprint arXiv:1312.3005*.

Doddington, G. (2002). Automatic evaluation of machine translation quality using n-gram co-occurrence statistics. In *Proceedings of the second international conference on Human Language Technology Research*, pages 138–145. Morgan Kaufmann Publishers Inc.

Khuc, V. N., Shivade, C., Ramnath, R., and Ramanathan, J. (2012). Towards building large-scale distributed systems for twitter sentiment analysis. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 459–464, New York, NY, USA. ACM.

Kirk, D. B. and Wen-mei, W. H. (2012). *Programming massively parallel processors: a hands-on approach*. Newnes.

Lin, J. (2009). Scalable language processing algorithms for the masses: A case study in computing word cooccurrence matrices with mapreduce. in proceedings of the conference on empirical methods in natural language processing. *EMNLP '08, pp. 419–428, Stroudsburg, PA, USA*.

Manning, C. D. and Schütze, H. (1999). *Foundations of statistical natural language processing*. MIT press.

McCormick, C. (2017). Word2vec tutorial part 2 - negative sampling [blog post]. `http://mccormickml.com/2017/01/11/word2vec-tutorial-part-2-negative-sampling/`.

Pang, B. and Lee, L. (2008). Opinion Mining and Sentiment Analysis. *Foundations and Trends® in Information Retrieval*, 2(12):1–135.

Pennington, J., Socher, R., and Manning, C. (2014). Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543.

Role, F. and Nadif, M. (2011). Handling the impact of low frequency events on co-occurrence based measures of word similarity - a case study of pointwise mutual information.

Schuetze, H. (1997). Document information retrieval using global word co-occurrence patterns. US Patent 5,675,819.

Weiss, S. M., Indurkhya, N., Zhang, T., and Damerau, F. J. (2005). From textual information to numerical vectors. In *Text Mining*, pages 15–46. Springer.