Implementação Paralela em C+CUDA de um Categorizador Multi-Rótulo de Texto Baseado no Algoritmo *k*-NN

Lucas Veronese, Alberto F. De Souza, Claudine Badue, Elias Oliveira
Universidade Federal do Espírito Santo, 29075-910, Vitória-ES
lucas.veronese@lcad.inf.ufes.br, alberto@lcad.inf.ufes.br, claudine@lcad.inf.ufes.br, elias@lcad.inf.ufes.br

Resumo

Em problemas de categorização automática de texto com um grande número de rótulos, as bases de dados de treinamento são grandes, o que pode tornar proibitivo o tempo de categorização para sistemas on-line. *Neste trabalho avaliamos a implementação paralela em C+CUDA* (Compute Unified Device Architecture) *de um categorizador multi-rótulo de texto baseado no algoritmo k-NN* (*k*-Nearest Neighbors). *Nós implementamos este algoritmo de duas formas: seqüencial em C e paralela em C+CUDA. Nossos resultados experimentais mostram que, com o uso de GPUs* (Graphics Processing Units) e *C+CUDA, são possíveis ganhos de desempenho da ordem de 65 vezes o desempenho seqüencial alcançado com CPUs.*

1. Introdução

Na categorização multi-rótulo de texto, um sistema tipicamente produz um conjunto de rótulos, cujo tamanho é desconhecido a priori, para cada documento sob análise. A maioria dos trabalhos em categorização automática de texto na literatura está focada em problemas de categorização uni-rótulo de texto, onde cada documento pode ter apenas um único rótulo [22]. Entretanto, em problemas do mundo real, a categorização multi-rótulo é freqüentemente necessária [21, 5, 1, 23, 6, 24]. Muitas técnicas de aprendizado de máquina têm sido usadas para construir sistemas automáticos para categorização de texto, tais como árvores de decisão [1], métodos de núcleo [5], redes neurais artificiais [2] ou métodos de aprendizagem preguiçosa [24], sendo muitas específicas para categorização multi-rótulo de texto [21, 23, 6, 2].

Para se obter bom desempenho de categorização, tipicamente são necessários muitos exemplares de treinamento para cada rótulo. Por essa razão, em problemas com um grande número de rótulos, as bases de dados de treinamento são grandes, o que pode tornar o tempo de categorização proibitivo para sistemas on-line. Neste trabalho, investigamos a implementação paralela em C+CUDA (*Compute Unified Device Architecture* [16]) de um categorizador multi-rótulo de texto baseado no algoritmo *k-Nearest Neighbors* (*k*-NN) [22]. Nossa motivação para a realização deste trabalho veio da necessidade de se implementar no país o Cadastro Sincronizado Nacional (CSN) de empresas [14].

O CSN integra as administrações tributárias federal, estaduais, municipais e demais órgãos envolvidos no processo de formalização das empresas, simplificando e racionalizando os procedimentos de abertura, manutenção e baixa de empresas. Uma das premissas do Cadastro é a coleta única de dados, desobrigando o cidadão a comparecer a vários órgãos para formalizar a sua empresa o que, por conseqüência, melhora o ambiente de negócios no país. Um dado fundamental que deve fazer parte do cadastro das empresas é um ou mais códigos que descrevam suas atividades econômicas segundo a Classificação Nacional de Atividades Econômicas (CNAE [10]) – a tabela CNAE atual (CNAE 2.0) lista as 1.300 atividades econômicas legalmente aceitas no país. Sempre que uma empresa é constituída ou tem seu cadastro alterado, seus códigos CNAE devem ser atribuídos ou revistos, respectivamente.

A automação da categorização de atividades econômicas de companhias, a partir de descrições destas atividades na forma de texto livre, é um grande desafio para a administração tributária. Atualmente, esta tarefa tem sido executada por funcionários públicos das três esferas de governo, nem todos apropriadamente treinados para esta tarefa. Além disso, quando o problema de categorização é resolvido diretamente por humanos, sua subjetividade traz um problema: diferentes categorizadores humanos podem atribuir diferentes categorias à uma mesma descrição de atividade econômica. Isto pode causar distorções na informação usada para planejamento, tributação e outras obrigações governamentais nos três níveis da administração: municipal, estadual e federal.

Estimamos que sejam necessárias cerca de 130.000 descrições de atividades econômicas com seus respectivos códigos CNAE para tornar possível a implementação de um categorizador automático com bom desempenho de categorização (cerca de 100 exemplares para cada código CNAE). Com esta quantidade de dados de treinamento, o tempo de categorização pode ser proibitivo para sistemas *on-line*. Vale destacar que a tarefa de atribuir ou revisar códigos CNAE de empresas é realizada no país cerca de 1.800.000 vezes por ano [4].

Para atacar este problema, examinamos a implementação paralela em C+CUDA de um categorizador de texto baseado no algoritmo *k*-NN. O categorizador *k*-NN é referência na literatura [22], sendo

usualmente empregado em comparações com outros categorizadores. Sua implementação paralela nos permitiria avaliar o potencial de *Graphics Processor Units* (GPUs) com arquitetura CUDA na solução eficiente deste importante problema.

CUDA foi desenvolvida dentro do escopo da indústria de GPUs. Diferente das unidades centrais de processamento (*Central Processing Units* – CPUs), que apresentaram uma melhoria de desempenho segundo padrão histórico nos últimos anos, o desempenho das GPUs cresceu enormemente (ver Figura 1 [17]).

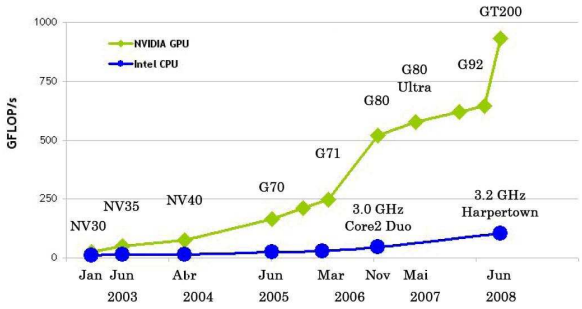

Figura 1: Evolução do desempenho: GPUs × CPUs [16].

A oportunidade de utilização da capacidade computacional de GPUs para execução de programas de propósito geral (*General Purpose Computation on GPUs* – GPGPU) tem sido explorada [11]; contudo, todos os mecanismos de programação de GPUs disponíveis até recentemente eram por demasiado voltados para facilitar o uso de GPUs para processamento gráfico. CUDA mudou este cenário por meio da disponibilização de um modelo de programação massivamente paralela que pode facilmente ser integrado ao modelo seqüencial de programação vigente – programas seqüenciais em C traduzidos para C+CUDA têm alcançado desempenhos em GPUs de dezenas a centenas de vezes superiores aos alcançados em CPUs [12].

Pesquisadores têm investigado o uso GPUs em diversos domínios de problemas, incluindo, entre outros, computação científica, banco de dados, busca na Web em larga escala, e categorização de texto [8, 19, 3, 7]. Entretanto, até onde conseguimos examinar, não existem na literatura trabalhos anteriores sobre o emprego de GPUs na categorização multi-rótulo de texto com grandes bases de documentos de treinamento.

Nós implementamos uma versão seqüencial em C e uma versão paralela em C+CUDA do algoritmo k-NN para categorização multi-rótulo de texto, sendo que esta última de duas formas distintas: (a) empregando bibliotecas CUDA pré-existentes; e (b) codificando todo o problema diretamente em C+CUDA. Com esta segunda versão paralela buscamos avaliar os ganhos da otimização manual do código. Nossos resultados experimentais mostram que são possíveis *speed-ups* superiores a 60 com o uso de C+CUDA otimizado manualmente.

Este artigo está organizado da seguinte forma. Após esta introdução, na Seção 2 formalizamos o problema de categorização multi-rótulo de texto e apresentamos o categorizador *k*-NN. Em seguida, na Seção 3, descrevemos sumariamente a arquitetura de GPUs CUDA. Na Seção 4 apresentamos a implementação paralela em C+CUDA de nossos categorizadores *k*-NN, na Seção 5 nossa metodologia para sua avaliação experimental, na Seção 6 os resultados dos experimentos e, finalmente, na Seção 7, nossas conclusões.

2. Categorização Multi-Rótulo de Texto

Seja $\mathbf{D}$ um domínio de documentos, $C = \{c_1, c_2, \ldots, c_{|C|}\}$ um conjunto pré-definido de categorias e $\Omega = \{d_1, d_2, \ldots, d_{|\Omega|}\}$ um *corpus* inicial de documentos previamente categorizados manualmente por peritos do domínio. Na categorização multi-rótulo, cada documento $d_j \in \Omega$ é categorizado em uma ou mais categorias de *C*.

Em um sistema de categorização baseado em aprendizado de máquina, Ω é dividido em dois subconjuntos, *TV* e *Te*. *TV* é usado para treinar (e validar eventuais parâmetros do) o sistema. Este treinamento é feito associando subconjuntos apropriados de *C* a características extraídas de cada documento $d_j \in TV$. *Te*, por outro lado, consiste de documentos para os quais as categorias apropriadas não são do conhecimento do sistema de categorização. Depois de ser treinado e validado com *TV*, o sistema de categorização é usado para predizer o conjunto de categorias de cada documento $d_j \in Te$.

Um sistema automático de categorização multi-rótulo tipicamente implementa uma função na forma $f : \mathbf{D} \times C \to \Re$ que retorna um número real que representa o grau de crença de cada par $\langle d_j, c_i \rangle \in \langle \mathbf{D} \times C \rangle$, isto é, um número entre 0 e 1 que representa a confiança do categorizador de que o documento de teste d_j deve ser categorizado sob a categoria c_i. A função $f(\cdot,\cdot)$ pode ser transformada numa função de *ranking* $r(\cdot,\cdot)$, tal que, se $f(d_j, c_i) > f(d_j, c_k)$, então $r(d_j, c_i) < r(d_j, c_k)$, e se $f(d_j, c_i) < f(d_j, c_k)$, então $r(d_j, c_i) > r(d_j, c_k)$.

Seja C_j o conjunto de categorias pertinentes ao documento de teste d_j. Um sistema de categorização bem sucedido tenderá a posicionar as categorias pertencentes a C_j em posições mais elevadas no *ranking* do que aquelas não pertencentes a C_j.

2.1. Indexação

Antes de serem categorizados, os textos devem ser convertidos para um formato apropriado para o categorizador empregado por meio de um procedimento denominado indexação [22] (a indexação é um mecanismo de extração de características). Para o categorizador *k*-NN, um texto d_j deve ser convertido em um vetor de pesos de termos $\vec{d}_j = \langle w_{1j}, \ldots, w_{|V|j} \rangle$, onde *V* é o conjunto de termos (ou palavras) que ocorrem pelo menos uma vez em um documento de *TV*, e $0 \le w_{kj} \le 1$

busca representar o quanto um termo t_k contribui para a categorização do documento d_j.

A função $tfidf(t_k, d_j)$ é a usualmente empregada no processo de indexação (*tfidf* significa *term frequency inverse document frequency*) [22]. $tfidf(t_k, d_j)$ retorna o peso w_{kj} do termo t_k no documento d_j, sendo dada por:

$$tfidf(t_k, d_j) = tf(t_k, d_j) \cdot \log \frac{|TV|}{df(t_k)} \quad (1)$$

onde $tf(t_k, d_j)$, ou *term frequency*, denota o número de vezes que t_k ocorre em d_j e $df(t_k)$, ou *document frequency*, denota o número de documentos em *TV* nos quais t_k ocorre. Para que os pesos estejam no intervalo [0, 1] e para que os documentos sejam representados por vetores de magnitude igual, os pesos computados por $tfidf(t_k, d_j)$ são geralmente normalizados conforme a Equação (2) [22].

$$w_{kj} = \frac{tfidf(t_k, d_j)}{\sqrt{\sum_{s=1}^{|V|} \left(tfidf(t_s, d_j)\right)^2}} \quad (2)$$

2.2. Categorizador *k-Nearest Neighbors* (*k*-NN)

O categorizador *k*-NN [22] encontra os *k* vizinhos mais próximos de um documento de entrada d_j no conjunto de documentos previamente aprendidos, *TV*, de acordo com alguma métrica de distância. Nos experimentos reportados neste artigo, usamos o cosseno do ângulo entre o vetor que representa d_j e o vetor que representa cada documento $d_i \in TV$. O cosseno é uma métrica de distância comumente usada na literatura [22] e dada por:

$$\cos(d_j, d_i) = \frac{\sum_{z=0}^{|V|} w_{jz} w_{iz}}{\sqrt{\sum_{z=0}^{|V|} w_{jz}^2} \sqrt{\sum_{z=0}^{|V|} w_{iz}^2}} \quad (3)$$

A função $f(d_j, c_k)$ do categorizador *k*-NN retorna o maior valor de $\cos(d_j, d_i)$ para $d_i \in TV$ e $c_k \in C_i$, onde C_i é o conjunto de categorias pertinentes ao documento d_i. O categorizador *k*-NN extrai os *k* pares $\langle d_j, c_i \rangle \in \langle \mathbf{D} \times C \rangle$ no topo do ranking construído a partir de $f(\cdot,\cdot)$.

O principal custo computacional do categorizador *k*-NN está relacionado ao cômputo do numerador da Equação (3) – um produto de matriz por vetor –, uma vez que, para se encontrar o maior $\cos(d_j, d_i)$ para $d_i \in TV$ e $c_k \in C_i$, todos os $d_i \in TV$ têm que ser examinados. O custo computacional do denominador é pequeno porque a norma de d_j só precisa ser computada uma vez para cada d_j (primeiro somatório no denominador) e as normas de $d_i \in TV$ (segundo somatório no denominador) podem ser computadas previamente e usadas para todos os d_j que se deseje categorizar.

3. *Compute Unified Device Architecture* (CUDA)

Para o programador de C+CUDA, a GPU é um co-processador da CPU capaz de executar dezenas de milhares de *threads* em paralelo. Trechos da aplicação com grandes demandas de computação podem ser traduzidos para CUDA e executados em GPUs. Para traduzir estes trechos para CUDA, o programador tipicamente reescreve sua versão seqüencial na forma de *kernels* paralelos, que podem ser funções simples ou aninhamentos complexos de funções.

Um *kernel* comanda a execução na GPU de um conjunto de *threads*, que são organizadas em grades (*grids*) de blocos de *threads* (*thread blocks*). Uma *grid* é um conjunto de *thread blocks* que executam independentemente, enquanto que um *thread block* é um conjunto de *threads* que podem cooperar por meio de sincronização do tipo barreira e acesso compartilhado a um espaço de memória exclusivo de cada *thread block*.

A Figura 2 mostra um trecho simples de código seqüencial em C e uma versão paralela do mesmo em C+CUDA. Na figura, __global__ indica que a função saxpy_paralelo é um *kernel* e deve ser executada na GPU. Em C+CUDA, *kernels* são invocados por meio de chamadas de função estendidas com o formato:

kernel <<<dimGrid, dimBlock>>> (parâmetros);

onde dimGrid e dimBlock são vetores de três elementos do tipo dim3 (parte da *Application Programming Interface* – API – de CUDA [16]) que especificam as dimensões das *grids* e *blocks*, respectivamente. Dimensões não especificadas são consideradas 1.

Computação serial de *y* ← *ax* + *y*, *y* e *x* vetores e *a* escalar

```
void saxpy_serial (int n, float a, float *x, float *y)
{
    for (int i = 0; i < n; i++)
        y[i] = a * x[i] + y[i];
}

// Invoca o kernel serial SAXPY
saxpy_serial (n, 2.0, x, y);
```

Computação paralela de *y* ← *ax* + *y* em C+CUDA

```
__global__
void saxpy_paralelo (int n, float a, float *x, float *y)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    if (i < n) y[i] = a * x[i] + y[i];
}

// Invoca o kernel paralelo SAXPY com 256 threads por bloco
int nblocks = (n + 255) / 256;
saxpy_paralelo <<<nblocks, 256>>> (n, 2.0, x, y);
```

Figura 2: Exemplo de código em C+CUDA.

No exemplo da Figura 2, o comando saxpy_paralelo <<<nblocks, 256>>> (n, 2.0, x, y) invoca uma *grid* unidimensional com nblocks *blocks* unidimensionais, cada um com 256 *threads* (por simplicidade, funções da API para alocação de memória e transferência de dados da CPU para a GPU e vice-versa não são mostradas). Cada uma destas *threads* computa um elemento do vetor resultado. Note que os valores de blockIdx.x,

blockDim.x, threadIdx.x, que identificam unicamente cada *thread*, são globais ao *kernel* e computados automaticamente durante a execução a partir dos parâmetros usados ao invocar o *kernel*. Note, também, que o código da Figura 2 funciona para qualquer n, limitado apenas pelo hardware da GPU.

GPUs hoje disponíveis possuem centenas de processadores e cada um pode executar *blocks* com até 512 *threads*, sendo o número de *blocks* por *grid* limitado apenas pela memória disponibilizada à GPU. Tipicamente, centenas *thread blocks* são possíveis, o que resulta em dezenas de milhares de *threads* por *kernel*. O custo de criar e chavear entre estas *threads* é extremamente baixo – poucos ciclos de relógio de GPU.

O grande número de *threads* por processador, centenas de processadores por GPU e o baixo custo do chaveamento entre *threads* (o que oculta a latência de memória) viabilizam o modelo de computação massivamente paralelo e de alto desempenho de C+CUDA.

A programação em C+CUDA é suportada por uma pequena extensão da linguagem C e por uma nova biblioteca C. A Figura 3 apresenta os níveis de abstração de um programa C+CUDA.

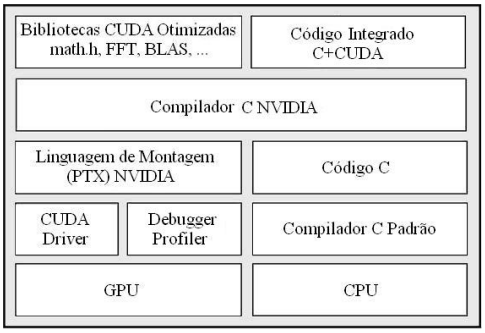

Figura 3: Níveis de abstração de um programa em C+CUDA [9].

Como mostra a Figura 3, no nível mais alto, um programa fonte em C+CUDA é especificado em C mais comandos específicos para a GPU, e pode incluir chamadas a funções das bibliotecas CUDA otimizadas da Nvidia. Um fonte em C+CUDA deve ser compilado pelo Compilador C da Nvidia, que gera código *assembly* (PTX [16]) para o *assembler* da Nvidia e código em C para ser compilado por um compilador C padrão (gcc). O código *assembly* traduzido para código de máquina de GPU é levado à GPU pelo *driver* da placa de vídeo CUDA *enabled* sob demanda do código de máquina de CPU produzido pelo compilador C.

Tanto a CPU (ou *host*) como a GPU (ou *device*) mantêm memória DRAM própria, chamadas de *host memory e device memory*, respectivamente. Dados podem ser copiados de forma otimizada de uma DRAM para outra através de chamadas à biblioteca CUDA [16].

Um programa em C+CUDA roda sua parte C no *host* e sua parte CUDA no *device*, sendo esta última formada por *grids* de *threads*, conforme mostrado na Figura 4. As *grids* são invocadas em seqüência de forma bloqueante ou não (mais de uma *grid* pode estar em execução ao mesmo tempo) e a função __syncthreads() implementa uma barreira para o sincronismo de *threads* de um mesmo bloco [16].

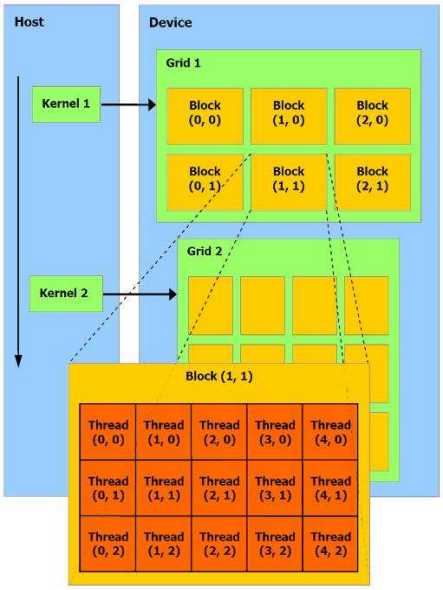

Figura 4: Hierarquia de *threads* em C+CUDA [16].

C+CUDA oferece ao programador autonomia para usar três tipos de memória presentes na GPU por meio de qualificadores do tipo das variáveis empregadas. O qualificador __device__ especifica que uma variável deve ser alocada na *global memory*, o qualificador __constant__ especifica que uma variável deve ser alocada na *constant memory*, e o qualificador __shared__ especifica que uma variável deve ser alocada na *shared memory*.

Uma variável automática definida em uma função *device* geralmente reside em um registrador (estas variáveis podem ser movidas para outras regiões de memória se não houver registradores suficientes). Os registradores são de acesso exclusivo de cada *thread*. A *shared memory* é compartilhada por todas as *threads* de um bloco e, apesar de pequena, é muito importante, pois possui baixa latência de acesso. A *constant memory* também possui baixa latência e pode ser acessada por todas as *threads* de uma *grid*, mas apenas para leitura – somente o *host* pode escrever nesta memória. A *global memory* é o local onde o *host* publica os dados que serão processados. Ela é compartilhada por todas as *threads* de uma *grid*, mas possui alta latência (400 a 600 ciclos de *clock* de GPU para uma leitura [16]). Existe outra memória voltada para aplicações gráficas, a *texture memory* (memória de textura), somente de leitura [16], que pode, contudo, ser usada por aplicações não gráficas.

4. Implementação Paralela do Categorizador *k*-NN em C+CUDA

No treinamento, o categorizador *k*-NN recebe e armazena uma matriz, *mTV*, de *documentos*×*termos*, e uma matriz, *mC*, de *documentos*×*categorias*. Um documento d_i descrevendo uma atividade econômica é, na verdade, a linha l_i da matriz *mTV* e seus códigos CNAE associados são aqueles da mesma linha l_i da matriz *mC*. Os pesos dos termos que ocorrem em cada documento d_i são as colunas da matriz *mTV*. O peso de um determinado termo t_k no documento d_i é computado de acordo com a função $tfidf(t_k, d_i)$ (ver Seção 2.1).

Na fase de teste do *k*-NN é necessário computar a distância entre um documento de teste d_j e cada documento d_i em *TV*. Essa distância é representada por $\cos(d_j, d_i)$. Como mencionado na Seção 2.2, a principal operação demandada pelo cálculo do $\cos(d_j, d_i)$ para cada documento d_i de *TV* é o produto da matriz *mTV* pelo vetor $\vec{d}_j$. Esta operação foi paralelizada por meio de um algoritmo implementado em C+CUDA.

Para compreender como paralelização do produto da matriz *mTV* pelo vetor $\vec{d}_j$ foi feita, suponha que *mTV* (*documentos*×*termos*) possui 7 linhas e 8 colunas. Para esta matriz 7×8 pode ser criada uma *grid* unidimensional com 4 blocos unidimensionais, cada um com 4 *threads*. Esta *grid* pode ser vista como uma matriz de *threads* de dimensões 4×4. A cada passo do algoritmo, as 4 *threads* de cada um dos blocos recebem cada uma um elemento de uma linha de *mTV* e o elemento correspondente de $\vec{d}_j$. A Figura 5 mostra como as *threads* da *grid* 4x4 são mapeadas aos 7×8 elementos de *mTV*. Na figura, o mapeamento das *threads* referente ao primeiro passo do algoritmo aparece em destaque.

(0,0)	(0,1)	(0,2)	(0,3)	(0,0)	(0,1)	(0,2)	(0,3)
(1,0)	(1,1)	(1,2)	(1,3)	(1,0)	(1,1)	(1,2)	(1,3)
(2,0)	(2,1)	(2,2)	(2,3)	(2,0)	(2,1)	(2,2)	(2,3)
(3,0)	(3,1)	(3,2)	(3,3)	(3,0)	(3,1)	(3,2)	(3,3)
(0,0)	(0,1)	(0,2)	(0,3)	(0,0)	(0,1)	(0,2)	(0,3)
(1,0)	(1,1)	(1,2)	(1,3)	(1,0)	(1,1)	(1,2)	(1,3)
(2,0)	(2,1)	(2,2)	(2,3)	(2,0)	(2,1)	(2,2)	(2,3)

Figura 5: Exemplo de como as *threads* fazem o produto de matriz por vetor.

A cada passo do algoritmo, a *grid* é deslocada para a direita de um número de colunas igual ao número de *threads* por bloco. Em cada um destes passos, cada *thread* computa o produto de um elemento de um documento $\vec{d}_i$ por um elemento correspondente de $\vec{d}_j$, e acumula este produto em uma variável denominada dj_di_product. Quando a *grid* é deslocada de modo a cobrir todos os elementos de uma linha, as variáveis dj_di_product são publicadas em um vetor de cada bloco de *threads*, denominado accum_dj_di_product, previamente alocado em memória compartilhada. O $\cos(d_j, d_i)$ de um documento d_j percorrido pela *grid* é igual à soma dos elementos do vetor accum_dj_di_product dividida pelo produto das normas $|d_j|$ e $|d_i|$. Os elementos do vetor accum_dj_di_product são somados por meio do algoritmo *Sum Tree Like Reduction* [20], o qual é descrito na Seção 6.

Após o cômputo de todos os $\cos(d_j, d_i)$ dos documentos d_j visitados pela *grid* no seu deslocamento para a direita, o mesmo é reposicionado à esquerda e deslocado para baixo de um número de linhas de *mTV* igual ao número de blocos de *threads*.

A Figura 6 mostra o código em C+CUDA de nosso algoritmo que deve ser executado por cada *thread*. No código da Figura 6, a variável mTV representa a matriz de treinamento do algoritmo de categorização, *mTV*. Na verdade, essa matriz foi implementada na forma de um vetor. A variável first guarda o índice para o início das linhas da matriz *mTV*, a variável last guarda o índice do final das linhas de *mTV*. O uso destas variáveis no código viabiliza o uso de uma *grid* com dimensões não necessariamente múltiplas das dimensões da matriz *mTV*.

```
__global__ void
matrix_vector_product(float *mTV, int num_line_mTV, int num_col_mTV, float *dj,
                      float *mTV_norm, float *cos_dj_di)
{
    __shared__ float accum_dj_di_product[NUMTHREAD];
    int first, last, i, j, k;
    float dj_di_product, norm_dj_di_product;

    for(i=blockIdx.x;i<num_line_mTV;i+=gridDim.x)
    {
        first=i*num_col_mTV;
        last=first+num_col_mTV;
        dj_di_product=0.0;
        k=threadIdx.x;

        for (j=threadIdx.x+first;j<last;j+=blockDim.x,k+=blockDim.x)
            dj_di_product+=dj[k]*mTV[j];
        accum_dj_di_product[threadIdx.x]=dj_di_product;
        __syncthreads();
        sum_tree_like_reduction(accum_dj_di_product,NUMTHREAD,&dj_di_product);
        if(threadIdx.x==0)
        {
            norm_dj_di_product=dj_norm*mTV_norm[i];
            if(norm_dj_di_product>0.000001)
                cos_dj_di[i]=dj_di_product/norm_dj_di_product;
        }
    }
}
```

Figura 6: Código do produto de matriz por vetor.

As variáveis num_line_mTV e num_col_mTV contêm o número de linhas e colunas de *mTV*, respectivamente; o vetor dj é o documento d_j que desejamos categorizar; mTV_norm guarda a norma $|d_i|$ de cada linha l_i de *mTV*; a variável dj_norm guarda a norma $|d_j|$; e o vetor cos_dj_di armazena $\cos(d_j, d_i)$ para cada linha de *mTV*.

As iterações do *loop* (for) mais externo deslocam a *grid* para baixo, enquanto que as iterações do *loop* (for) mais interno: (i) deslocam a *grid* para a esquerda, (ii) computam os produtos de cada elemento de $\vec{d}_i$ por cada elemento correspondente de $\vec{d}_j$, e (iii) acumulam estes produtos na variável dj_di_product de cada *thread*.

Depois de cada *thread* percorrer as posições e fazer o produto, elas colocam o resultado no vetor accum_dj_di_product. Cada *thread* fica responsável pelo cômputo de uma posição deste vetor: a *thread* 0 pela posição 0, a *thread* 1 pela posição 1, e assim sucessivamente. Finalmente, os elementos do vetor

accum_dj_di_product são somados, e a *thread* 0 calcula a divisão desta soma pelo produto da norma $|d_j|$ e pela norma $|d_i|$. Como mencionado anteriormente, para somar os elementos de accum_dj_di_product foi usado o algoritmo *Sum Tree Like Reduction*, descrito a seguir.

4.1. Sum Tree Like Reduction

O algoritmo *Sum Tree Like Reduction* tem como propósito somar todos os elementos de um vetor e pode ser facilmente paralelizado em C+CUDA.

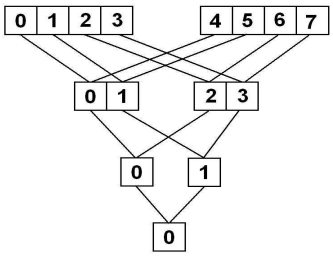

Figura 7: Exemplo da soma em árvore (*Sum Tree Like Reduction*).

Suponha que o vetor cujos elementos queremos somar tenha dimensão 8. Para somar seus elementos, os mesmos são divididos em dois grupos (ver Figura 7). São somados os elementos 0 e 4 e o resultado é colocado em 0; 1 e 5 e o resultado é colocado em 1; 2 e 6 e o resultado é colocado em 2; e por fim, são somados os elementos 3 e 7 e o resultado é colocado em 3. Em seguida, o vetor é reduzido para tamanho 4 e novamente é dividido em dois. Assim, são somados os elementos 0 e 2 e o resultado é colocado em 0; e 1 e 3 e o resultado é colocado em 1. Finalmente, os elementos onde foram colocados os resultados das últimas somas são divididos novamente em dois grupos e são somados os elementos 0 e 1, e o resultado é colocado em 0. O resultado da soma de todos os elementos do vetor fica no elemento 0 ao final da execução do algoritmo.

```
__device__ void
sum_tree_like_reduction(float *vector, int vector_size, float *sum)
{
    int j, k;

    for(k=vector_size/2; k>0; k>>=1)
    {
      __syncthreads();
      for(j=threadIdx.x; j<k; j+=blockDim.x)
         vector[j]+=vector[k+j];
    }
    __syncthreads();
    if(threadIdx.x==0)
       *sum=vector[0];
}
```

Figura 8: Código da soma em árvore (*Sum Tree Like Reduction*).

A Figura 8 mostra o código paralelo em C+CUDA do algoritmo *Sum Tree Like Reduction*. Os parâmetros de entrada do algoritmo são: o vetor que desejamos somar, vector; o tamanho deste vetor, vector_size; e a variável onde o resultado será retornado, sum. É importante lembrar que o tamanho do vetor deve ser potência de dois. No final do cômputo da soma, a *thread* 0 será a única que terá o valor da soma. As iterações do *loop* (for) mais externo fazem a divisão do vetor que desejamos somar, e as iterações do *loop* (for) mais interno são responsáveis por fazer a soma em paralelo.

5. Metodologia

Nós implementamos uma versão seqüencial e duas versões paralelas do algoritmo *k*-NN: uma conforme descrito na Seção 4, e uma que faz uso da biblioteca CUBLAS [18] para o cômputo do produto matriz por vetor. O uso de bibliotecas pré-existentes facilita o desenvolvimento de código, mas, como veremos na Seção 6, pode não permitir extrair o melhor do desempenho hardware.

Nós comparamos experimentalmente os desempenhos em termos de tempo das três versões do algoritmo *k*-NN implementadas. Os experimentos de comparação foram conduzidos da seguinte forma.

Separamos um documento, d_j, do nosso conjunto de dados Ω para compor nosso conjunto de teste *Te* (*Te* só possui um elemento), e particionamos o restante de Ω em 10 subconjuntos disjuntos de tamanho igual para realizarmos 10 experimentos de medida de desempenho. No primeiro experimento, treinamos nossos categorizadores (seqüencial e paralelos) com o primeiro subconjunto (*TV* igual a 10% de Ω) e medimos o tempo de categorização de d_j com os três categorizadores; no segundo, treinamos os categorizadores com dois subconjuntos (*TV* igual a 20% de Ω) e medimos novamente o tempo de categorização de d_j; e nos 8 demais experimentos seguimos o mesmo procedimento.

A métrica utilizada para comparar o desempenho dos categorizadores foi o *speed-up*, que é a razão entre o tempo de categorização seqüencial e o paralelo. No cálculo do *speed-up* foi utilizada a média do tempo de 100 execuções de cada categorizador para reduzir o impacto de variáveis alheias ao desempenho dos algoritmos (carga do sistema operacional, etc.).

5.1. Base de Dados

A base de dados empregada em nossa avaliação experimental compreende descrições textuais de atividades econômicas de companhias brasileiras categorizadas em um subconjunto de códigos CNAE por funcionários públicos treinados nesta tarefa. O conjunto de dados Ω empregado consiste de 6.911 descrições de atividades econômicas de 6.911 empresas de Vitória-ES ou Belo Horizonte-MG, categorizadas em 105 diferentes atividades econômicas (categorias). Cada uma destas categorias ocorre em exatamente 100 documentos diferentes deste conjunto de dados, i.e., existem 100 instâncias de documentos para cada categoria.

5.2. Pré-Processamento dos Dados

Nós transformamos todas as palavras em nosso conjunto de dados em suas formas não flexionadas

(termos), i.e., a forma da palavra que aparece no dicionário (conhecida como lema [13]), e então removemos todas as preposições usando o dicionário eletrônico Diadorim para o Português do Brasil [15]. Em seguida, identificamos todos os termos distintos presentes em Ω, i.e., o vocabulário de interesse V (3.764 termos). Finalmente, transformamos todos os documentos dos conjuntos de treinamento e teste em seus vetores multi-dimensionais de pesos de termos correspondentes, $\vec{d}_i = \langle w_{1i}, \ldots, w_{|V|i} \rangle$ e $\vec{d}_i = \langle w_{1i}, \ldots, w_{|V|i} \rangle$, respectivamente (ver Seção 2.1). Vale mencionar que os categorizadores avaliados neste artigo não possuem nenhum parâmetro a ser ajustado, não sendo necessária a fase de validação.

5.3. CPU e GPU

Os experimentos foram executados em uma CPU AMD Athlon 64 X2 (*Dual Core*) 5.200+ de 2,7 GHz, com 512KB de cache L2 por *core* e 3GB de DRAM DDR2 de 800 MHz. O Sistema Operacional empregado foi o Linux Fedora 9 e o compilador C o gcc 4.3.0.

A placa de vídeo utilizada foi uma NVIDIA GeForce GTX 285 com 1GB de DRAM GDDR3. A versão do compilador CUDA foi o nvcc 2.1. A GPU da GTX 285 possui vários *Stream Processors* (SPs) para executar operações inteiras e de ponto flutuante. Os SPs são agrupados em *Stream Multiprocessors* (SMs) que, por sua vez, são agrupados em *Thread Processing Clusters* (TPCs [17]). A GTX 285 possui 10 TPCs de 3 SMs, cada um com 8 SPs, totalizando 240 SPs, e opera com um *clock* de 1,48 GHz. A memória principal de 1GB possui *clock* de 2.484 MHz, barramento de 512 *bits* e a taxa de transferência de 159 GB/s.

A GTX 285 permite criar 512 *threads* por bloco e é capaz de manter o estado de 32K *threads* simultaneamente (uma *grid* de 32K *threads*). Ela pode processar dados em precisão simples e dupla, mas o desempenho máximo em precisão dupla é de 80 Gflop/s, enquanto que o em precisão simples é igual à 933 Gflop/s [17].

6. Resultados Experimentais

A Tabela 1 apresenta, para cada tamanho do conjunto de treinamento *TV*, as médias dos 100 tempos de categorização (em segundos) de um documento dos categorizadores seqüencial e paralelos (N. Alg. significa Nosso Algoritmo). Ela também apresenta os *speed-ups* obtidos com o uso dos algoritmos paralelos.

Como a Tabela 1 mostra, os *speed-ups* crescem com o tamanho de *TV* e, com os tamanhos empregados, *speed-ups* de até cerca de 65 vezes podem ser obtidos com nossa versão do *k*-NN.

A Figura 9 apresenta os resultados mostrados na Tabela 1 de forma gráfica. Na figura, o eixo *x* representa o tamanho de *TV* e o eixo *y* o *speed-up*. Nela pode-se observar que, a partir de 2764 documentos de treinamento, o *speed-up* começa a estabilizar em ambas as versões paralelas. Isto ocorre porque as partes seqüenciais, quais sejam, a cópia do documento que desejamos categorizar da memória da CPU para a memória da GPU, a criação das *threads* na GPU e a cópia dos resultados de volta para a memória da CPU, passam a ser menos relevantes no tempo total dos algoritmos paralelos. Como a Figura 9 mostra, nosso algoritmo apresenta desempenhos consistentemente superiores que a versão paralela implementada com a CUBLAS. Nós acreditamos que este resultado se deve ao fato de que a função para computar o produto matriz por vetor da CUBLAS realiza operações adicionais que tipicamente ocorrem quando se deseja computar o produto matriz por vetor [18], mas que não ocorrem no nosso caso.

Tabela 1: Média e *speed-up* do tempo de categorização de um documento para os algoritmos seqüencial e paralelos.

|*TV*|	Tempo seqüencial (s)	Tempo CUBLAS (s)	*Speed-up* CUBLAS	Tempo N. Alg. (s)	*Speed-up* N. Alg.
691	0,02096	0,00069	30,2	0,00047	44,3
1382	0,04159	0,00119	34,9	0,00078	53,3
2073	0,06259	0,00167	37,5	0,00107	58,5
2764	0,08305	0,00168	49,3	0,00133	62,3
3455	0,10067	0,00218	46,2	0,00163	61,7
4146	0,11825	0,00266	44,5	0,00190	62,1
4837	0,13787	0,00313	44,1	0,00219	63,1
5528	0,15758	0,00317	49,8	0,00250	63,0
6219	0,17719	0,00364	48,7	0,00274	64,6
6910	**0,19602**	**0,00413**	**47,5**	**0,00302**	**64,8**

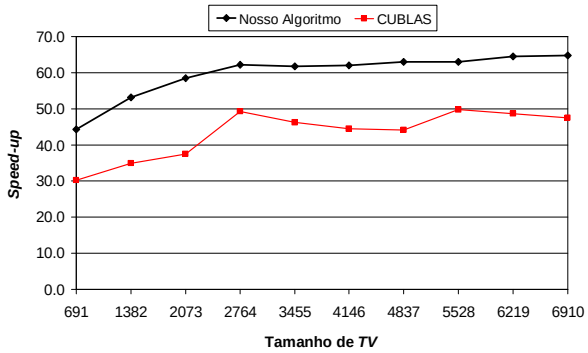

Figura 9: Gráfico de *speed-up* do algoritmo *k*-NN.

7. Conclusões

Neste trabalho nós examinamos os benefícios do uso de GPUs na categorização automática de texto multi-rótulo. Quando a base de documentos de treinamento dos categorizadores é grande, o tempo de categorização pode crescer a ponto de inviabilizar a categorização *on-line* de documentos. Por meio da implementação paralela dos categorizadores em C+CUDA e sua execução em GPUs CUDA *enabled*, é possível aumentar o desempenho em termos de tempo de categorização e viabilizar solução *on-line* deste importante problema computacional.

Nós implementamos duas versões paralelas em C+CUDA de um categorizador multi-rótulo *k*-NN e comparamos seu desempenho com o de uma versão seqüencial. Nossos resultados experimentais mostraram

que *speed-ups* da ordem de 65 vezes podem ser obtidos. O tempo seqüencial de categorização de nosso algoritmo *k*-NN para uma base de treinamento de 6.910 documentos em um computador pessoal topo de linha aproxima-se do limite admissível para sistemas *on-line* (0,2 segundos – ver Tabela 1). Para o caso real de categorização de atividades econômicas segundo a CNAE, acreditamos que seja necessário treinar categorizadores com cerca de 130.000 documentos. É verdade que outros algoritmos de categorização multi-rótulo possam ter complexidade de tempo menor que o *k*-NN; contudo, os *speed-ups* obtidos mostram o potencial de sistemas baseados em GPU na área de Recuperação de Informação.

8. Referências

[1] F. D. Comité, R. Gilleron, and M. Tommasi, "Learning Multilabel Alternating Decision Tree from Texts and Data", *Lecture Notes in Computer Science 2734*, 2003, pp. 35–49.

[2] A. F. De Souza, F. Pedroni, E. Oliveira, P. M. Ciarelli, W. F. Henrique, L. Veronese, and C. Badue, "Automated Multi-label Text Categorization with VG-RAM Weightless Neural Networks", *Neurocomputing 72(10-12)*, 2009, pp. 2209-2217.

[3] S. Ding, J. He, H. Yah, and T. Suel, "Using Graphics Processors for High Performance IR Query Processing", *18th Iinternational Conference on World Wide Web (WWW'08)*, 2008, pp. 421-430.

[4] DNRC, "Ranking das Juntas Comerciais segundo Movimento de Constituição, Alteração e Extinção e Cancelamento de Empresas ", *Departamento Nacional de Registro do Comércio (DNRC)*, http://www.dnrc.gov.br/Estatisticas/ranking_2008.htm, último acesso em 09/07/2009.

[5] A. Elisseeff and J. Weston, "A Kernel Method for Multilabelled Classification", *Advances in Neural Information Processing Systems 14*, 2002, pp. 681–687.

[6] S. Gao, W. Wu, C.-H. Lee, and T.-S. Chua, "A MFoM Learning Approach to Robust Multiclass Multi-label Text Categorization", *Proceedings of the 21st International Conference on Machine Learning*, 2004, pp. 329–336.

[7] V. Garcia, E. Debreuve, and M. Barlaud, "Fast k Nearest Neighbor Search Using GPU", *Computer Vision and Pattern Recognition Workshops*, 2008, pp. 1 – 6.

[8] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha, "GPUTeraSort: High Performance Graphics Co-Processor Sorting for Large Database Management", *26th ACM SIGMOD International Conference on Management of Data*, 2006, pp. 325-336.

[9] T. R. Halfhill, "Parallel Processing with CUDA: Nvidia's High-Performance Computing Platform Uses Massive Multithreading", *Microprocessor*, January 2008.

[10] IBGE, "Classificação Nacional de Atividades Econômicas - Fiscal (CNAE-Fiscal)", *Instituto Brasileiro de Geografia e Estatística (IBGE)*, Rio de Janeiro, RJ, 2003.

[11] D. Luebke, M. Harris, J. Krüger, T. Purcell, N. Govindaraju, I. Buck, C. Woolley, A. Lefohn, "GPGPU: general purpose computation on graphics hardware", *International Conference on Computer Graphics and Interactive Techniques, ACM SIGGRAPH 2004, Course Notes*, 2004.

[12] D. Luebke, "GPU Computing: The Democratization of Parallel Computing", 13th International Conference on *Architectural Support for Programming Languages and Operating Systems (ASPLOS'08), Course Notes, 2008.*

[13] C. D. Manning, P. Raghavan, and H. Schütze, "An Introduction to Information Retrieval", Cambridge University Press, Cambridge, England, 2008.

[14] MF, "Cadastro Sincronizado Nacional (CSN)", *Ministério da Fazenda*, https://www16.receita.fazenda.gov.br/CadSinc/, último acesso em 06/07/2009.

[15] NILC, "Diadorim: A Lexical Database for Brazilian Portuguese", *Núcleo Interinstitucional de Lingüística Computacional (NILC)*, http://www.nilc.icmc.usp.br/nilc/tools/intermed.htm, último acesso em 06/07/2009.

[16] NVIDIA, "NVIDIA CUDA: Compute Unified Device Architecture - Programming Guide 2.0", 2008.

[17] NVIDIA, "Technical Brief: NVIDIA GeForce GTX 200 GPU Architectural Overview", 2008.

[18] NVIDIA, "CUDA CUBLAS Library", 2008.

[19] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. Purcell, "A Survey of General-Purpose Computation on Graphics Hardware", *Computer Graphics Forum 26(1)*, 2007, pp. 80-113.

[20] J. Owens, "Data-Parallel Algorithms and Data Structures", *Tutorial on* http://gpgpu.org/feed *High Performance Computing with CUDA, International Conference for High Performance Computing, Networking, Storage and Analysis*, http://gpgpu.org/static/sc2007/SC07_CUDA_4_DataParallel_Owens.pdf, último acesso em 09/07/2009.

[21] R. E. Schapire and Y. Singer, "BoosTexter: a Boosting-Based System for Text Categorization", *Machine Learning 39(2/3)*, 2000, pp. 135–168.

[22] F. Sebastiani, "Machine Learning in Automated Text Categorization", *ACM Computing Surveys 34(1)*, 2002, pp. 1-47.

[23] N. Ueda and K. Saito, "Parametric Mixture Models for Multilabel Text", *Advances in Neural Information Processing Systems 15*, 2003, pp. 721–728.

[24] M.-L. Zhang and Z.-H. Zhou, "ML-kNN: A Lazy Learning Approach to Multi-Label Learning", *Pattern Recognition 40(7)*, 2007, pp. 2038–2048.