# Implementação Paralela em C+CUDA de uma Rede Neural Probabilística

Lucas Veronese, Alberto F. De Souza, Claudine Badue, Elias Oliveira
*Laboratório de Computação de Alto Desempenho, Departamento de Informática*
*Universidade Federal do Espírito Santo, Av. Fernando Ferrari 514, 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

*Com a evolução e o grande poder computacional das Graphics Processing Units (GPUs), elas passaram a ser usadas para vários propósitos. Em 2006, a NVIDIA lançou a arquitetura CUDA (Compute Unified Device Architecture). A partir disso, o uso de GPUs para processamento não gráfico ganhou força na comunidade científica. Neste trabalho, investigamos uma implementação paralela em C+CUDA de uma Rede Neural Artificial Probabilística (*Probabilistic Neural Network - *PNN) empregada na categorização multi-rotulada de documentos textuais.*

## 1. Introdução

Na categorização automática multi-rotulada de documentos textuais, 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 nós implementamos uma versão seqüencial em C e uma versão paralela em C+CUDA do algoritmo PNN para categorizar textos. Nossos resultados experimentais mostram que são possíveis ganhos de desempenho superiores a 70 vezes o desempenho seqüencial com o uso de C+CUDA.

O Cadastro Sincronizado Nacional (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 [3]) – 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.

A programação em C+CUDA é viabilizada por uma pequena extensão da linguagem C e por uma nova biblioteca C. A GPU (*device*) é vista pela CPU (*host*) como um co-processador capaz de executar um número muito grande de *thread*s em paralelo. Tanto o *host* como o *device* mantêm memória (*Dynamic Random Access Memory* - DRAM) própria, chamadas de *host memory e device memory*. Os dados podem ser copiados de forma otimizada de uma DRAM para a outra através de chamadas à biblioteca CUDA [5].

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* [5].

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 [4][6][8][9]. 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.

Este trabalho está organizado da seguinte forma. Na seção 2 formalizamos o problema de categorização de texto. Em seguida, na seção 3 descrevemos o algoritmo PNN e implementação dele em C+CUDA. Na seção 4 apresentamos base de dados utilizada nos experimentos e também avaliamos os resultados. Finalmente, na Seção 5, a conclusões e trabalhos futuros.

## 2. Categorização

Seja **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 $d_i \in \Omega$ 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 de) o sistema. Este treinamento é feito associando subconjuntos apropriados de *C* a características extraídas de cada documento $d_i \in TV$. *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 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 por meio de um procedimento denominado indexação [7]. Para o categorizador PNN, um texto $d_j$ deve ser convertido em um vetor de pesos de termos $\vec{d}_j = \langle w_{1,j}, \ldots, w_{|V|,j} \rangle$, onde *V* é o conjunto de termos (ou palavras) que ocorrem pelo menos uma vez em um documento de *TV*, e $w_{k,j}$ busca representar o quanto um termo $t_k$ contribui para a categorização do documento $d_j$.

## 3. Rede Neural Probabilística

A Rede Neural Probabilística (*Probabilistic Neural Network – PNN*) proposta por Oliveira et. al [1] é capaz de resolver problemas de categorização de texto com múltiplos rótulos. Esta PNN é composta por três camadas: camada de entrada, de padrões e de soma (Figura 1).

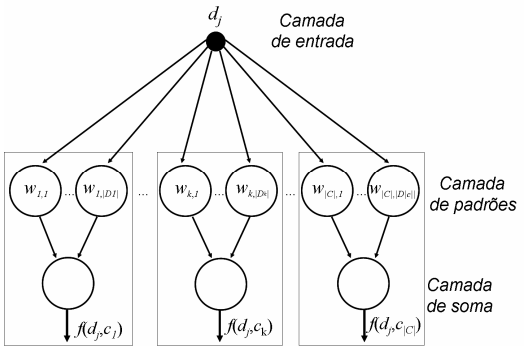


**Figura 1: Arquitetura da PNN.**

No treinamento, a PNN recebe e armazena uma matriz, *mTV*, de *documentos×termos*, e uma matriz, *mC*, de *documentos×categorias*. O documento $d_i$ é a linha *i* da matriz *mTV* e as categorias associadas a ele são aquelas da linha *i* da matriz *mC*. Os pesos dos termos que ocorrem em cada documento $d_i$ estão nas colunas da matriz *mTV*. Cada coluna de *mC* representa, então, uma das categorias de *C*.

Para cada documento $d_i$ da matriz *mTV* é criado um conjunto de neurônios, um para cada categoria $c_k \in C_i$, onde cada neurônio $n_{k,i}$ armazena o vetor $d_i$ com um vetor de pesos de termos, $w_{k,i}$.

Na fase de categorização é apresentado à camada de entrada um documento $d_j$, o qual se deseja classificar. Nessa camada nada é computado, ela simplesmente passa o documento $d_j$ para a camada de padrões. Cada neurônio $n_{k,i}$ da camada de padrões possui a função de ativação $A(d_j, c_k, n_i)$, apresentada na Equação 1:

$$A(d_j, c_k, n_i) = \frac{1}{2\pi\sigma} \exp\left(\frac{w_{k,i} \cdot d_j - 1}{\sigma^2}\right), \quad k=1, \ldots, |C|, i=1, \ldots, |D_k| \qquad (1)$$

onde $\sigma$ é constante para todos os neurônios (ajustada no treino para o melhor desempenho de categorização), $C$ é conjunto de categorias possíveis, e $D_k$ é o conjunto de documentos associados a categoria $c_k$.

A principal operação realizada pelo algoritmo é o produto interno $w_{k,i} \cdot d_j$. Portanto, nós concentramos nossos esforços no paralelismo desta operação em C+CUDA. Apesar de um documento de treino $d_i$ resultar na criação de mais de um neurônio na camada de padrões, i.e., um para cada $c_k$, o produto $w_{k,i} \cdot d_j$ só precisa ser computado uma vez para cada $d_i$. Ou seja, todos os produtos $w_{k,i} \cdot d_j$ necessários podem ser obtidos por meio do produto matriz por vetor $mTV_{d_j}$.

Para computar o produto $mTV_{d_j}$ são criados 128 blocos unidimensionais de 256 *threads* unidimensionais para percorrer a matriz *mTV*, de dimensões 6910 (documentos) x 3764 (termos), fazendo o produto.

O produto matriz por vetor é feito da seguinte forma. 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 $d_i$ por um elemento correspondente de $d_j$, 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 produto interno $w_{k,i} \cdot d_j$ é igual à soma dos elementos do vetor accum_dj_di_product. A função $A(d_j, c_k, n_i)$ é computada com estas somas.

Após o cômputo de todos os produtos internos $w_{k,i} \cdot d_j$ 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 2 mostra o código em C+CUDA de nosso algoritmo, que deve ser executado por cada *thread*.

No código da Figura 2, 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*.

O próximo passo é o cômputo da camada onde são feitas as somas dos resultados da camada padrões. Na camada de soma, que tem $|C|$ neurônios, cada neurônio está associado com uma categoria $c_k$ e computa a função $f(d_x,c_k)$, que é apresentada na Equação 2.

$$f(d_x, c_k) = \sum_{i=1}^{|N_k|} A(d_x, c_k, n_i), \; k=1, ..., |C| \qquad (2)$$

onde $N_k$ é o número de neurônios da camada de padrões associados a $c_k$.

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

  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;
    j=threadIdx.x;

    for(j=+first;j<last;j+=blockDim.x){
       dj_di_product+=dj[k]*mTV[j];
       k+=blockDim.x
    }
    accum_dj_di_product[threadIdx.x]=dj_di_product;
    sum(accum_dj_di_product,NUMTHREAD);
    __syncthreads();

    if(threadIdx.x==0){
        r=(accum_dj_di_product[0]-1)/powf(sigma,2);
        A_dj_di[i]=powf(2*PI*sigma,-1)*expf(r);
    }
  }
}
```

**Figura 2: Código do produto de matriz por vetor.**

## 4. Experimentos

Nesta seção, nós apresentaremos os resultados experimentais da paralisação do algoritmo PNN. Fizemos uma comparação de desempenho, em termos de tempo, do algoritmo implementado seqüencial, em C padrão, e a versão paralela em C+CUDA, a ser executada na GPU.

### 4.1. Estrutura do Experimento

Os experimentos foram executados em uma CPU AMD Athlon 64 X2 (Dual Core) 5.400+ de 2,8 GHz, com 512KB de cache L2 por core e 4GB de DRAM DDR2 de 800 MHz. A placa de vídeo utilizada foi uma NVIDIA GeForce GTX 285.

Nosso conjunto de dados, chamado de EX100, é composto de 6911 documentos (descrições textuais) contendo 105 atividades econômicas diferentes (categorias). Cada uma dessas categorias ocorre exatamente em 100 diferentes documentos neste conjunto, isto é, existem 100 exemplares de documentos de cada categoria. O número médio de categorias por documento é rigorosamente 1,52 (desvio padrão de 0,79). As características de EX100 permitem

avaliar o desempenho dos classificadores no caso onde as classes (ou rótulos) são uniformemente distribuídas em todo o conjunto.

### 4.2. Eficiência de Tempo

Nós comparamos experimentalmente os desempenhos em termos de tempo das duas versões do algoritmo PNN 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 de igual para realizarmos 10 experimentos de medida de desempenho. No primeiro experimento, treinamos nossos classificadores (seqüencial e paralelo) com o primeiro subconjunto (*TV* igual a 10% de Ω) e medimos o tempo de categorização de $d_j$ com os dois 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.

### 4.3. Análise dos Resultados

A Tabela 1 apresenta as médias dos tempos (em segundos) de 100 execuções do algoritmo PNN, seqüencial e paralelo, para cada tamanho de *TV*. A variância foi menor que 1.0e-7 e o desvio padrão que 1.0e-3. Ela também apresenta o *speed-up* alcançado pelo algoritmo paralelo executando na GPU. Como a Tabela 1 mostra, o *speed-up* cresce com o tamanho de *TV*. A nossa versão paralela do PNN chegou a 72 vezes mais rápido do que a seqüencial.

**Tabela 1: Média dos tempos de categorização de um documento para os algoritmos seqüencial e paralelos.**

| \|*TV*\| | C (s) | C+CUDA (s) | *Speed-up* |
|---|---|---|---|
| 691 | 0,01963 | 0,00055 | 35,7 |
| 1382 | 0,03824 | 0,00085 | 45,1 |
| 2073 | 0,05684 | 0,00107 | 52,9 |
| 2764 | 0,07541 | 0,00128 | 58,8 |
| 3455 | 0,09398 | 0,00152 | 61,8 |
| 4146 | 0,11258 | 0,00174 | 64,8 |
| 4837 | 0,13117 | 0,00193 | 67,9 |
| 5528 | 0,14976 | 0,00216 | 69,4 |
| 6219 | 0,16833 | 0,00237 | 71,1 |
| **6910** | **0,18694** | **0,00259** | **72,3** |

Nesta tabela pode-se observar que, o *speed-up* não aumenta de forma linear. A partir 5528 documentos de treinamento os *speed-ups*, são bem próximo, começando a estabilizar. Isto ocorre porque as partes seqüenciais, por exemplo, a cópia do documento de teste da memória da CPU para a da GPU, a criação dos *threads* na chamada do categorizador que executa na GPU, a cópia do resultado de categorização da memória da GPU para CPU, passaram a influenciar menos no tempo total do algoritmo paralelo.

## 5. Conclusões

Neste trabalho avaliamos os benefícios do uso de GPUs para a categorização automática multi-rotulada de documentos. Quando o número de documentos de treinamento é grande, o tempo para a categorização de pode inviabilizar a categorização *on-line* de documentos. Sendo assim, com uma implementação paralela de uma Rede Neural Artificial Probabilística em C+CUDA, nós comparamos o ganho de desempenho que pode ser obtido com o algoritmo rodando em GPU GeForce GTX 285, em relação a um algoritmo implementado na linguagem C padrão.

Com as análises dos resultados pudemos constatar que as GPUs são boas alternativas para computação paralela, logo usá-las em categorização de texto têm uma grande vantagem, pois muitos algoritmos são paralelizáveis. No entanto, as GPUs têm uma limitação de memória e as bases de dados normalmente usadas são de grandes dimensões. Assim, como trabalho futuro nós pretendemos utilizar uma forma de compactação de dados para que possamos usar base de dados maiores.

## 6. Referências


[1] E. Oliveira,P. M. Ciarelli,A. F. De Souza,and C. Badue. Using a Probabilistic Neural Network for a Large Multi-Label Problem. *Proceedings of the 10th Brazilian Symposium on Neural Networks (SBRN'08)*, pp. 195-200, Salvador, Bahia, Brazil, October 2008.

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

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

[4] 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.

[5] NVIDIA. NVIDIA CUDA: Compute Unified Device Architecture - Programming Guide 2.0. 2008a.

[6] 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.

[7] SEBASTIANI, F. Machine Learning in Automated Text Categorization. ACM Computing Surveys, 34(1): p. 1-47, 2002.

[8] 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.

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