Implementação de Redes Neurais em Plataforma GPU

Francisco Machado Magalhães Neto
Dep. de Ciência da Computação
Universidade Federal de Lavras
Lavras - MG, Brasil, 37200-000
Email: franmagneto@gmail.com

Wilian Soares Lacerda
Dep. de Ciência da Computação
Universidade Federal de Lavras
Lavras - MG, Brasil, 37200-000
Email: lacerda@dcc.ufla.br

Vancley Oliveira Simão
Dep. de Ciência da Computação
Universidade Federal de Lavras
Lavras - MG, Brasil, 37200-000
Email: vancleys@gmail.com

***Resumo*—O presente trabalho propõe a melhoria do desempenho de Redes Neurais Artificiais, beneficiando-se de seu alto grau de paralelismo ao implementá-las no modelo de computação paralela GPGPU (*General Purpose Graphics Processing Unit*), utilizando a tecnologia CUDA (*Compute Unified Device Architecture*), da empresa NVIDIA. A implementação utiliza dois níveis de paralelismo (de exemplos e de nó), e executa com todos os dados necessários já carregados para a memória da GPU de antemão, evitando transferências constantes, que seriam um gargalo no desempenho. O trabalho atingiu seu objetivo, demonstrando através dos resultados o ganho de desempenho obtido com esta implementação.**

***Palavras chave*—Redes Neurais Artificiais, Computação Paralela, GPU, GPGPU, CUDA.**

I. Introdução

As Redes Neurais Artificiais (RNA) são modelos computacionais baseados no funcionamento do cérebro humano, simulando neurônios e a comunicação entre eles [1]. As aplicações desse modelo são muitas, entre elas, reconhecimento de padrões, mineração de dados, reconhecimento de imagens e classificação.

Existem diversos modelos de RNAs e cada um pode ser treinado de determinada forma para a aplicação em dado problema. As redes neurais mais difundidas são as do tipo MLP (*Multilayer Perceptron*), que são uma generalização do modelo *Perceptron* [1]. Como os cálculos associados a cada neurônio são independentes, as redes neurais são paralelas por natureza [2].

As GPUs foram inicialmente concebidas para a tarefa exclusiva de processar elementos gráficos para exibição em dispositivos de apresentação. Conforme foram sendo aperfeiçoadas para processamento de gráficos mais elaborados (cenas em três dimensões, por exemplo), e cada vez mais complexos, as GPUs atingiram um alto poder de processamento.

Em função de sua alta capacidade de processamento paralelo, e custo relativamente baixo, observou-se que a GPU poderia ser usada como alternativa para a implementação de aplicações paralelas que exigem alta capacidade de processamento, mas que não necessariamente envolvam gráficos. Essa abordagem é chamada GPGPU (*General Purpose Computation on Graphics Processing Units*) [3].

Visando esse tipo de aplicação, a *NVIDIA Corporation*, empresa fabricante de GPUs criou a plataforma CUDA (*Compute Unified Device Architecture*), que baseia-se em uma extensão da linguagem de programação C e consiste em um compilador e uma API (*Application Programming Interface*) de programação. Esta plataforma gera programas para serem executados em GPUs NVIDIA.

A. Motivação

Como em qualquer solução computacional, é altamente útil e desejável melhorar o desempenho das RNAs, principalmente quando a rede em questão possui alta complexidade, com muitos neurônios e sinapses, e também quando o conjunto de dados utilizado é muito grande, contendo muitas amostras e/ou muitos atributos. Explorando a característica de independência dos cálculos em uma RNA, é possível melhorar o seu desempenho ao implementá-las de forma paralela.

Considerando que as GPUs são dispositivos de custo relativamente baixo, e alta capacidade de processamento, tornam-se uma boa alternativa para implementação de algoritmos que podem tirar vantagem do seu modelo de processamento paralelo, entre eles, a execução e treinamento de uma RNA.

B. Objetivos

O objetivo geral do presente trabalho é melhorar o desempenho (tempo de execução) do treinamento de uma Rede Neural, implementando-a de maneira paralela. Para atingir o objetivo geral acima proposto, foram definidos os seguintes objetivos específicos:

- Verificar a viabilidade da implementação da rede neural MLP em plataforma CUDA para execução em uma GPU NVIDIA.
- Comparar o desempenho de uma rede implementada dessa forma com uma implementação sequencial executada em CPU.

II. Redes Neurais Artificiais

As RNAs (*Redes Neurais Artificiais*) são modelos de aprendizagem de máquina inspirados no funcionamento das redes de neurônios biológicos, como as encontradas no cérebro humano. Há uma motivação em explorar a forma como o cérebro processa as informações, por esta ser *complexa*, *não-linear* e *paralela* [1].

De forma geral, uma *rede neural* é uma máquina projetada para modelar a forma como o cérebro realiza algumas funções; normalmente é implementada por componentes eletrônicos, ou simulada por programação em um computador digital [1].

A. O neurônio artificial

No trabalho de *W. Mcculloch & W. Pitts* [4], foi proposto um modelo matemático do fluxo de sinais em uma rede de neurônios. Foi o primeiro modelo conhecido de neurônio artificial.

Bernard Widrow & Marcian Hoff propuseram em 1959 modelos de neurônios chamados de *ADALINE* e *MADALINE*[1]. O modelo ADALINE foi desenvolvido para reconhecer padrões binários, de forma a poder prever o próximo bit em uma leitura de uma linha telefônica, com base nas leituras anteriores [5].

Segundo *S. S. Haykin* [1], o neurônio artificial apresenta três elementos básicos:

1) Um conjunto de *sinapses*, cada uma caracterizada por um peso. Uma entrada x_j na sinapse j conectada ao neurônio k é multiplicada pelo peso sináptico w_{kj}.
2) Um *somador*, responsável por somar todos os sinais de entrada, ponderados pelo respectivo valor de peso.
3) Uma *função de ativação* para restringir a amplitude do sinal de saída do neurônio.

O peso da sinapse $x_0 = +1$ representa o valor do *bias*, responsável por aumentar ou diminuir a entrada líquida da função de ativação. O modelo apresentado pode ser descrito matematicamente segundo a equação 1.

$$v_k = \sum_{j=0}^{m} w_{kj} x_j y_k = \phi(v_k) \tag{1}$$

B. Perceptron

F. Rosenblatt [6] propôs um modelo em que os neurônios eram organizados em camadas de entradas e saídas, e os pesos sinápticos da soma ponderada das entradas eram ajustáveis. Dessa forma, era possível treinar a rede ajustando os valores dos pesos sinápticos, configurando assim um sistema de *aprendizagem*. Tal modelo foi chamado de *Perceptron*.

A aprendizagem do *Perceptron* se dá por correção de erro, através da aplicação da regra da equação 2[2]:

$$w(n+1) = w(n) + \eta[d(n) - y(n)]x(n) \tag{2}$$

Onde $x(n)$ representa a entrada, $y(n)$ a saída, $d(n)$ a resposta desejada, a diferença $d(n) - y(n)$ representa o *erro*, e η é a *taxa de aprendizagem* [1].

C. Multilayer Perceptron

O *perceptron* simples descrito anteriormente só é capaz de resolver problemas linearmente separáveis, conforme apontado por *W. Mcculloch & W. Pitts* [7]. O modelo *Multilayer Perceptron*, chamado em diante de MLP, consiste no arranjo de uma rede perceptron com propagação para frente (*feed-forward*) dividida em múltiplas camadas, nas quais as saídas $y(n)$ de uma camada se tornam entradas da camada seguinte. A primeira camada é a *camada de entrada*, a última é a *camada de saída*, e as intermediárias são denominadas *camadas escondidas*. A figura 1 retrata o modelo MLP.

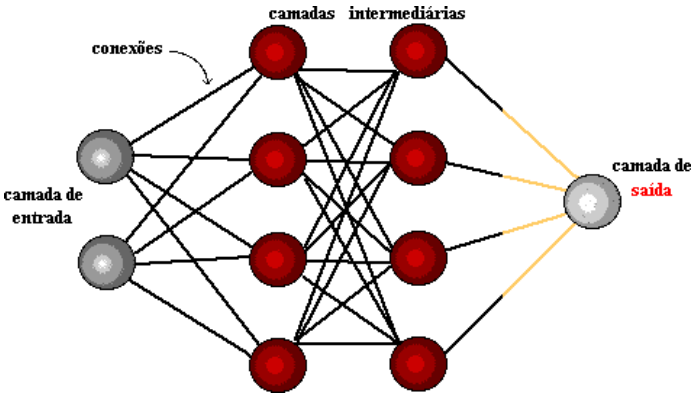

Figura 1. Organização em camadas. Fonte: [8]

Em geral, a função de ativação dos neurônios das camadas escondidas em uma MLP é uma *função sigmóide* (em forma de s), sendo a *função logística*, apresentada na equação 3 uma das funções desse tipo mais comuns utilizadas para este fim [1].

$$\phi(v) = \frac{1}{1 + \exp(-av)} \tag{3}$$

D. Backpropagation

Ainda não havia sido criado um algoritmo eficiente para treinamento de redes MLP, até que *D. E. Rumelhart, G. E. Hinton, & R. J. Williams* [9] propuseram o *Backpropagation*. Assim, tornou-se possível aplicar RNAs a problemas não linearmente separáveis, utilizando MLPs com treinamento *backpropagation*.

S. S. Haykin [1] apresenta o algoritmo da seguinte forma:

- Propagar os sinais de entrada através de todas as camadas da rede, *para frente*, e computar o sinal de erro.
- *Retropropagação*. Cálculo dos gradientes locais δs.
- Ajuste dos pesos segundo a *Regra Delta Generalizada* (equação 4)

$$w_{ji}^{(l)}(n+1) = w_{ji}^{(l)}(n) + \alpha[w_{ji}^{(l)}(n-1)] + \eta\delta_j^{(l)}(n) y_i^{(l-1)}(n) \tag{4}$$

III. General-Purpose computation on Graphics Processing Units

GPGPU significa *General-Purpose computation on Graphics Processing Units* (Computação de Propósito Geral em Unidades de Processamento Gráfico), também conhecido como *Computação em GPU* [10].

As GPUs (*Graphics Processing Units*) são dispositivos projetados para aceleração do processamento de elementos

[1]Multiple ADAptive LINear Elements.
[2]As vezes referida como *regra delta*.

gráficos para exibição em monitores de vídeo. As primeiras GPUs de fato eram dedicadas somente a essas tarefas, suportando apenas pipelines de função fixa específicos [3]. As GPUs atuais são implementadas como *stream processors*, e são programáveis.

Pesquisadores passaram a tirar proveito do desempenho de ponto flutuante destas GPUs, usando-as para computação genérica [3]. O termo GPGPU foi cunhado por Mark Harris em 2002, quando fundou o site *gpgpu.org* ao reconhecer a nova tendência de usar GPUs para aplicações não gráficas [10].

A. Computação Paralela

Um computador paralelo é um conjunto de processadores capazes de trabalhar cooperativamente para resolver um problema computacional. Isso se aplica a supercomputadores, redes de estações de trabalho, ou estações e trabalho com múltiplos processadores [11].

B. Stream Processors

> O modelo de programação *stream* expõe a localidade e a concorrência em aplicações de processamento de mídia [12].

Segundo *S. Rixner* [12], no modelo *stream*, as aplicações são codificadas como uma sequência de *kernels* que operam em fluxos de dados. Um *kernel* é definido como um pequeno programa que é repetido para cada elemento do fluxo de entrada para produzir um fluxo de saída. Tal fluxo alimenta *kernels* subsequentes.

Uma arquitetura *stream* pode explorar a concorrência no processamento de mídia, quando as aplicações são expressas no modelo de programação *stream*. Os *kernels* assim expressos podem ser escalonados para execução em *clusters* com várias unidades aritméticas. Tais unidades são organizadas em um modelo SIMD[3], que executa operações idênticas em elementos diferentes dos dados [12].

IV. CUDA – *Compute Unified Device Architecture*

Em 2006, a NVIDIA apresentou a plataforma CUDA (*Compute Unified Device Architecture*), desenvolvida por Ian Buck, que havia se juntado à empresa. A plataforma consiste numa combinação de *hardware* e *software* [3]:

- **Hardware:** As GPUs da NVIDIA que vem com a tecnologia.
- **Software:** Uma extensão da linguagem C, com uma API de programação, chamada *CUDA C*.

Existem três abstrações chave na plataforma: *hierarquia de grupos de threads*, *memórias compartilhadas* e *sincronização por barreira*. A linguagem *CUDA C* permite definir rotinas como *kernels*, para serem executadas na arquitetura *stream* da GPU. Um *kernel* é chamado passando a quantidade N de *threads* a executá-lo simultaneamente.

Um *kernel* pode ser executado por múltiplos blocos, desde que tenham a mesma forma. Os blocos são organizados em um *grid* de uma, duas ou três dimensões, conforme mostra a figura 2 [13].

[3] *Single Instruction, Multiple Data*.

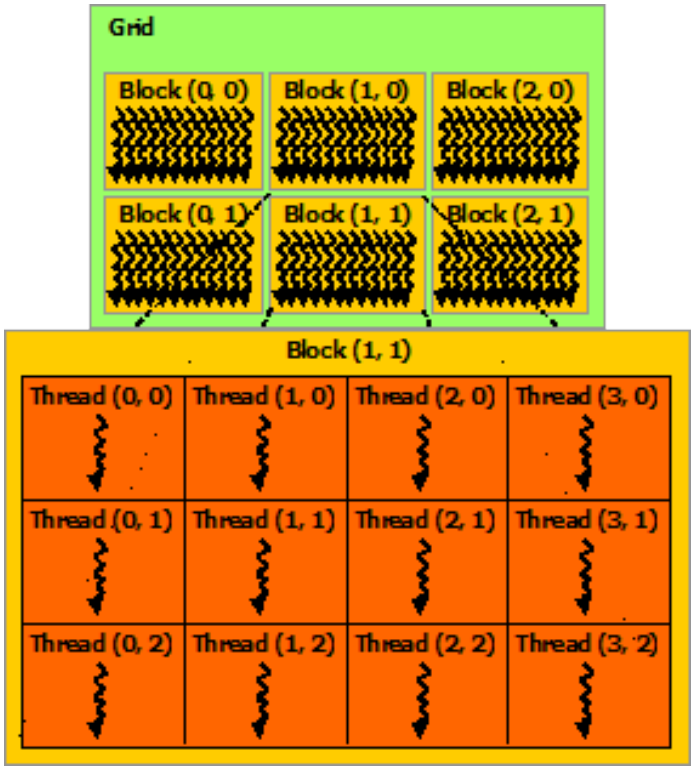

Figura 2. *Grid* de blocos de *threads*. Fonte: [13]

V. Implementação paralela de Redes Neurais Artificiais

Existem diversas técnicas para paralelização de Redes Neurais. O trabalho de *K. Aggarwal* [2] cita as seguintes:

- Paralelismo nas sessões de treinamento (executando várias sessões simultaneamente)
- Aprendizado simultâneo
- Paralelismo de camadas (execução concorrente de camadas)
- Paralelismo de nó (neurônio)
- Paralelismo dos pesos (cálculo simultâneo de matrizes de pesos)

O artigo de *S. Gurgel & A. De A Formiga* [14] apresenta uma implementação de Redes Neurais do tipo MLP em *CUDA*[4], utilizando dois níveis de paralelismo:

- Paralelismo de exemplos: alocando cada exemplo de entrada a um bloco de threads
- Paralelismo de nó: alocando o processamento de cada neurônio a uma thread

VI. Materiais e Métodos

Esta seção apresenta os procedimentos metodológicos de execução do trabalho. Foram implementados algoritmos para execução e treinamento de uma rede neural do tipo MLP, de maneira paralela, para execução em GPU, e de maneira sequencial, para execução em CPU.

[4] Implementação disponível no endereço https://github.com/NeuroGPU/neurogpu.

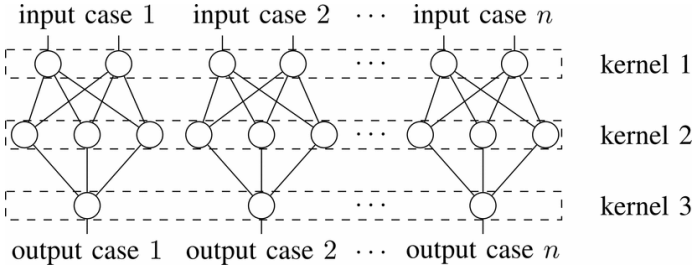

Figura 3. Funcionamento do programa neurogpu. Fonte: [14]

A implementação foi adaptada do programa *neurogpu* [14], codificado na linguagem *CUDA C* na versão paralela, e *C* na versão sequencial. Esta implementação possui paralelismo de exemplos e de nós. Não apresenta paralelismo de camadas: para cada camada da rede, um *kernel* é lançado, utilizando a quantidade de exemplos de entrada como número de blocos, e a quantidade de neurônios da camada como número de threads. Em seguida, o programa passa para a próxima camada, utilizando as saídas do kernel anterior como entradas. O processo é repetido até chegar à última camada.

O programa original foi adaptado para permitir a execução de testes com qualquer conjunto de testes com dados numéricos em formato *CSV*, definindo os parâmetros para o teste em um arquivo texto. Também foi incluso um contador de tempo de treinamento, e a possibilidade de alterar a quantidade de casos de entrada processados por um bloco de *threads*, e a quantidade de neurônios processados por uma única *thread*, possibilitando assim variar a quantidade de blocos e de *threads* a executar o treinamento. Um esquema do funcionamento do processo é mostrado na figura 3.

O programa foi executado em um computador com processador *Intel(R) Core(TM)2 E8400* com frequência de *clock* de 3 GHz, 4 GB de memória DDR2 a 667 MHz, e uma GPU *NVIDIA GeForce GTX 285*, com 240 núcleos *CUDA* e frequência de *clock* de 1.476 MHz, com 1 GB de memória GDDR3 a 1.242 MHz. O sistema operacional instalado é o *Arch Linux*, com a plataforma *CUDA* versão 6.5.

O tempo de execução do treinamento das RNAs pelo algoritmo *backpropagation* foi medido e comparado com uma implementação sequencial em CPU, utilizando 4 bases de dados distintas.

A. Dados utilizados

As bases de dados utilizadas para os testes são apresentados na tabela I.

As bases *cancer*, *car* e *spambase* foram retiradas do repositório de aprendizado de máquina da Universidade de Irvine [15], e representam problemas de classificação. Foi desenvolvido um pequeno programa em *CUDA* para gerar os dados de todas as bases *sin*, que representam problemas de aproximação de função.

B. Aquisição e geração de dados

Foram obtidos os dados de exemplos do site da Universidade de Irvine [15], e foi escrito o programa *sin-gen* em *CUDA* que gera os valores de entrada e calcula o seno para cada um em paralelo, de acordo com o número de amostras passado como parâmetro. Foram geradas bases com 1.000, 10.000, 100.000 e 1.000.000 de amostras da função seno.

Tabela I
DADOS UTILIZADOS

Descrição	Amostras	Entradas	Nome
Classificação de tumores	55	9	*cancer*
Classificação de carros	1.728	6	*car*
Classificação de spam	4.601	57	*spam*
Aproximação da função seno	1.251	1	*sin1k*
Aproximação da função seno	12.501	1	*sin10k*
Aproximação da função seno	125.001	1	*sin100k*
Aproximação da função seno	1.250.001	1	*sin1m*

C. Adaptação

O programa *neurogpu* foi adaptado e generalizado para utilização com bases de dados diferentes, utilizando um arquivo de descrição contendo os parâmetros de configuração da rede e dos dados para cada teste. As bases de dados foram formatadas e normalizadas para facilitar seu uso no programa. Foram incluídos contadores para medir tempo de treinamento da rede: uma função `StartTimer`, que faz a leitura do tempo atual do sistema é chamada antes da rotina de treinamento da rede (`BatchTrainBackprop`), e logo em seguida a função `StopTimer`, que faz a leitura novamente é chamada. A função `GetElapsedTime` retorna o tempo decorrido entre as chamadas das funções `StartTimer` e `StopTimer`. Também foi feita uma adaptação para permitir a variação no número de *threads* e blocos de execução na GPU.

D. Teste das implementações

Ambas as implementações foram executadas (CPU e GPU) tendo como entrada os mesmos conjuntos de dados. A versão GPU foi executada várias vezes com variação na quantidade de *threads* por bloco, e na quantidade de blocos[5], variando-se o número de entradas processadas por cada bloco, e o número de neurônios processados por cada *thread*.

Os parâmetros das redes neurais foram obtidos empiricamente, e os pesos inciais para cada rede foram fixados, garantindo o mesmo resultado na execução da RNA, para que o tempo de treinamento com a mesma evolução dos pesos sinápticos pudesse ser comparado nas diferentes execuções. Cada execução foi repetida dez vezes e os tempos de execução do treinamento foram coletados. Os testes são feitos com números altos de épocas de treinamento, para facilitar a observação das diferenças de tempo.

E. Comparação dos resultados

Foram coletados os dados da etapa anterior, e foi calculada a média e desvio-padrão para cada teste. Foi calculado o *speedup* (ganho de desempenho) em cada teste, dividindo o tempo

[5]Exceto para as bases da função seno com mais de mil amostras.

Tabela II
RESULTADO DOS TESTES

Dados	Tempo CPU (s)	Tempo GPU (s)	*Speedup*
cancer	0,239±0,003338	0,1564±0,001492	1,528
car	11,54±0,06299	2,216±0,004395	5,205
spambase	163,79±0,8364	15,37±0,03072	10,656
sin1k	7,056±0,0324	1,387±0,000449	5,087
sin10k	71,85±0,771	12,84±0,01365	5,597
sin100k	715,64±4,28	102,84±0,01866	6,959
sin1m	7252,88±216,85	1028,39±0,06765	7,053

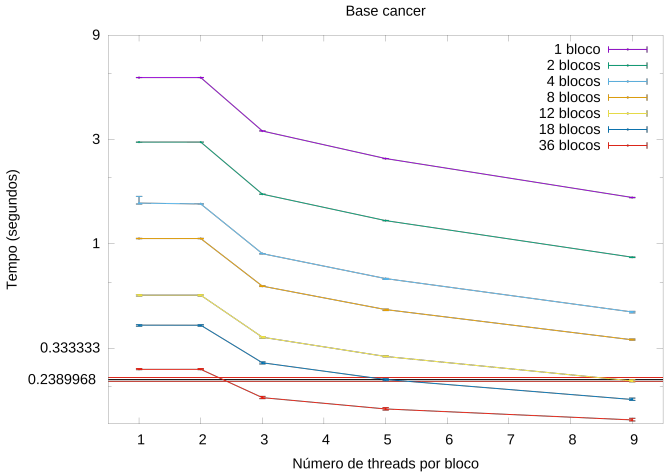

Figura 4. Classificação de tumores.

médio de treinamento da versão sequencial pelo da versão paralela.

VII. RESULTADOS

O tempo médio de cada teste, em CPU e GPU, com os desvios-padrão e *speedup* são apresentados na tabela II.

A. *Testes com alteração no número de threads*

Os testes com variação no número de blocos e *threads* na GPU foram executados com as bases *cancer* (figura 4), *car* (figura 5), *sin1k* (figura 6) e *spam* (figura 7).

Em cada gráfico, uma curva representa o tempo de execução do treinamento da RNA para uma dada quantidade de blocos de *threads*, e cada ponto representa a variação neste tempo para um dado número de *threads* por bloco. Os testes não apresentaram um desvio padrão significativo, portanto na maioria dos gráficos, os limites superior e inferior[6] não são visíveis.

Os gráficos também apresentam três linhas representando os tempos de execução do mesmo teste em CPU, sendo a linha central na cor preta, representando o tempo médio, e as outras duas na cor vermelha, mostrando os tempos máximo e mínimo.

[6]Maior e menor tempo em 10 execuções.

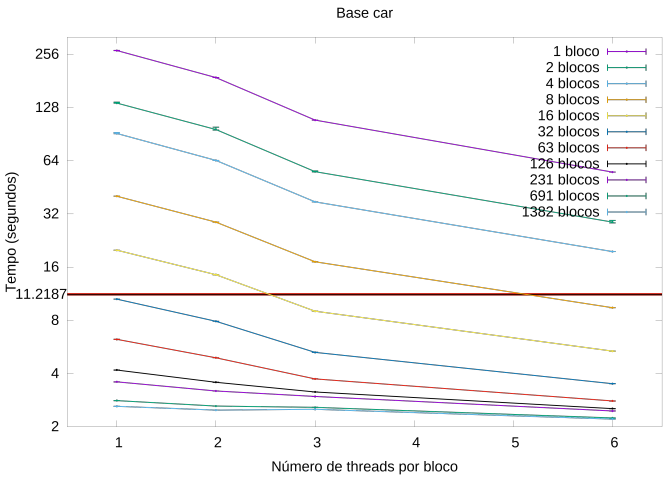

Figura 5. Classificação de carros.

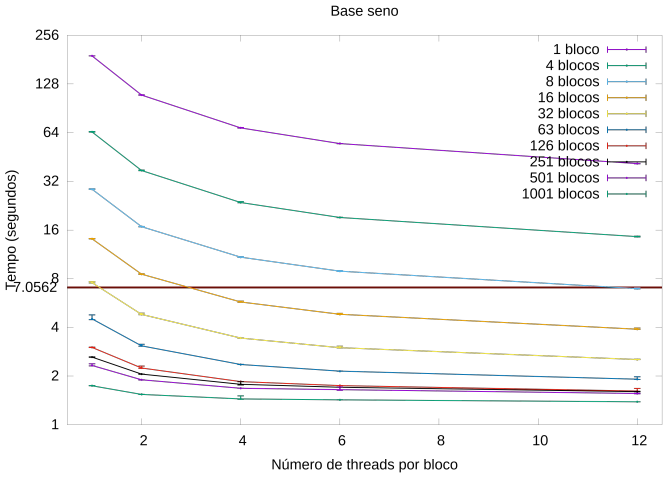

Figura 6. Aproximação da função seno com 1000 amostras.

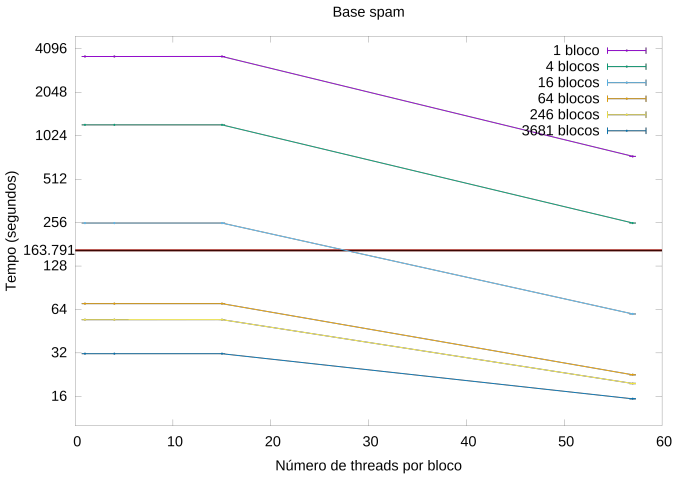

Figura 7. Classificação de *spam*.

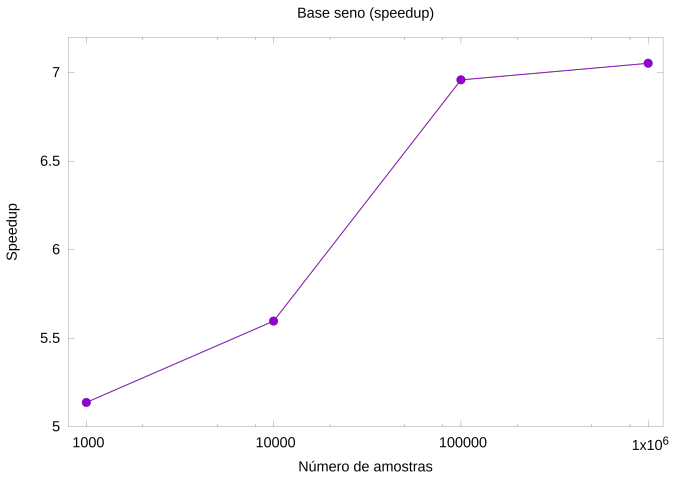

Figura 8. *Speedup* da aproximação da função seno variando o número de amostras.

B. Comparação dos resultados

No teste com a base de dados *carros*, com 1.728 instâncias de 6 atributos cada, foi observado um *speedup* de 5,2 da versão em GPU em relação à versão sequencial em CPU.

A base *spam* possui 4.601 instâncias, cada uma com 57 atributos, e é a maior base entre os problemas de classificação. Também foi a base que obteve maior *speedup*: 10,66.

A base *cancer* é a menor de todas, com 55 instâncias, e obteve o menor *speedup* (1,5) da versão GPU em relação à versão CPU.

Nos testes com a função seno, o *speedup* variou de 5,09 a 7,05, conforme a variação do tamanho da amostra. Entre os tamanhos de 1.000 e 10.000, foram obtidos *speedups* de 5,09 e 5,6. Entre 10.000 e 100.000 houve a maior variação no *speedup*, que aumentou para 6,96. Entre 100.000 e 1.000.000, o *speedup* foi de 7,05. A evolução do *speedup* nos testes de aproximação de função é mostrada no gráfico da figura 8.

Os testes com variação no número de blocos e de *threads* revelam um tempo de execução maior do que a versão em CPU para quantidades pequenas de blocos e de *threads*. É possível observar claramente a diminuição do tempo de execução tanto com o aumento de blocos quanto de *threads* por bloco.

VIII. Conclusão

A adaptação da implementação *neurogpu* foi bem-sucedida, e os testes mostram o ganho de desempenho obtido com a paralelização em GPGPU, que é tanto maior quanto for a quantidade de blocos e *threads* utilizados. O ganho de desempenho também tende a ser maior conforme a quantidade de dados de entrada, e de atributos destes.

A. Sugestões de trabalhos futuros

A GPU disponível para a implementação do projeto só é suportada até a versão 6.5 da plataforma CUDA, já que possui poder de computação nível 1.3. Propõe-se a melhoria da implementação utilizando recursos mais recentes da plataforma, em GPUs suportadas.

A plataforma CUDA é restrita às GPUs do fabricante Nvidia. Propõe-se uma reimplementação do trabalho em uma plataforma que suporte *hardware* de diversos fabricantes[7].

Sugere-se também a inclusão de um módulo de pré-processamento, capaz de realizar a leitura de dados em diversos formatos (rótulos de atributos e classe em forma de texto, por exemplo), e realize as adaptações necessárias para o funcionamento da rede (conversão de rótulos em números, normalização).

Propõe-se ainda a comparação do desempenho entre outras implementações de Redes Neurais em GPU.

Referências

[1] S. S. Haykin, *Redes Neurais*, 2nd ed. BOOKMAN COMPANHIA ED, 2001. [Online]. Available: https://encrypted.google.com/books?id=lBp0X5qfyjUC

[2] K. Aggarwal, "Simulation of artificial neural networks on parallel computer architectures," in *Educational and Information Technology (ICEIT), 2010 International Conference on*, vol. 2, 2010, pp. V2–255–V2–258.

[3] "Plataforma de computação paralela," NVIDIA Corporation, 2012. [Online]. Available: http://www.nvidia.com.br/object/cuda_home_new_br.html

[4] W. Mcculloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *Bulletin of Mathematical Biology*, vol. 5, no. 4, pp. 115–133, Dec. 1943. [Online]. Available: http://dx.doi.org/10.1007/BF02478259

[5] E. Roberts, "Neural networks," 2000. [Online]. Available: http://cs.stanford.edu/people/eroberts/courses/soco/projects/neural-networks/History/history1.html

[6] F. Rosenblatt, "The perceptron: a probabilistic model for information storage and organization in the brain." *Psychological review*, vol. 65, no. 6, pp. 386–408, Nov. 1958. [Online]. Available: http://psycnet.apa.org/journals/rev/65/6/386/

[7] M. Minsky and S. A. Papert, "Perceptrons: An introduction to computational geometry," *IEEE Transactions on Information Theory*, 1969.

[8] C. Y. Tatibana and D. Y. Kaetsu, "Aplicações de redes neurais artificiais," 2009. [Online]. Available: http://www.din.uem.br/ia/neurais

[9] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," *Parallel Distributed Processing*, vol. 1, 1986.

[10] "General-purpose computation on graphics hardware," GPGPU.org, 2002. [Online]. Available: http://gpgpu.org/about

[11] I. Foster, *Designing and Building Parallel Programs*. Addison-Wesley, 1995. [Online]. Available: http://www.mcs.anl.gov/~itf/dbpp

[12] S. Rixner, *Stream processor architecture*. Norwell, MA, USA: Kluwer Academic Publishers, 2002.

[13] *CUDA C Programming Guide 5.5*, NVIDIA Corporation, 2013. [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[14] S. Gurgel and A. De A Formiga, "Parallel implementation of feed-forward neural networks on gpus," in *Intelligent Systems (BRACIS), 2013 Brazilian Conference on*, Oct 2013, pp. 143–149.

[15] M. Lichman, "UCI machine learning repository," 2013. [Online]. Available: http://archive.ics.uci.edu/ml

[7] A biblioteca OpenCL, por exemplo.