UNIVERSIDADE FEDERAL DA PARAÍBA
CENTRO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

ANÁLISE DE TÉCNICAS DE IMPLEMENTAÇÃO PARALELA PARA TREINAMENTO DE REDES NEURAIS EM GPU

SÁSKYA THEREZA ALVES GURGEL

JOÃO PESSOA-PB
Janeiro-2014

UNIVERSIDADE FEDERAL DA PARAÍBA
CENTRO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

ANÁLISE DE TÉCNICAS DE IMPLEMENTAÇÃO PARALELA PARA TREINAMENTO DE REDES NEURAIS EM GPU

SÁSKYA THEREZA ALVES GURGEL

JOÃO PESSOA-PB
Janeiro-2014

SÁSKYA THEREZA ALVES GURGEL

ANÁLISE DE TÉCNICAS DE IMPLEMENTAÇÃO PARALELA PARA TREINAMENTO DE REDES NEURAIS EM GPU

DISSERTAÇÃO APRESENTADA AO CENTRO DE INFORMÁTICA DA UNIVERSIDADE FEDERAL DA PARAÍBA, COMO REQUISITO PARCIAL PARA OBTENÇÃO DO TÍTULO DE MESTRE EM INFORMÁTICA (SISTEMAS DE COMPUTAÇÃO).

Orientador: Prof. Dr. Andrei de Araújo Formiga

JOÃO PESSOA-PB
Janeiro-2014

G979a Gurgel, Sáskya Thereza Alves.
Análise de técnicas de implementação paralela para treinamento de redes neurais em GPU / Sáskya Thereza Alves Gurgel.- João Pessoa, 2014.
70f. : il.
Orientador: Andrei de Araújo Formiga
Dissertação (Mestrado) - UFPB/CI
1. Informática. 2. Sistemas de computação. 3. Redes neurais. 4. Computação paralela. 5. GPU.

UFPB/BC CDU: 004(043)

Ata da Sessão Pública de Defesa de Dissertação de Mestrado de **SASKYA THEREZA ALVES GURGEL**, candidata ao Título de Mestre em Informática na Área de Sistemas de Computação, realizada em 31 de janeiro de 2014.

Ao trigésimo primeiro dia do mês de janeiro do ano dois mil e quatorze, às quatorze horas, no auditório do CCEN - Universidade Federal da Paraíba - reuniram-se os membros da Banca Examinadora constituída para examinar o candidato ao grau de Mestre em Informática, na área de "*Sistemas de Computação*", na linha de pesquisa "*Computação Distribuída*", a Sra. **SASKYA THEREZA ALVES GURGEL**. A comissão examinadora foi composta pelos professores doutores: ANDREI DE ARAUJO FORMIGA (PPGI-UFPB), Orientador e Presidente da Banca, THAIS GAUDENCIO DO REGO (PPGI-UFPB), examinadora interna e AQUILES MEDEIROS FILGUEIRA BURLAMAQUI (UFRN), como examinador externo. Dando início aos trabalhos, o professor ANDREI DE ARAUJO FORMIGA cumprimentou os presentes, comunicou aos mesmos a finalidade da reunião e passou a palavra ao candidato para que o mesmo fizesse, oralmente, a exposição do trabalho de dissertação intitulado "*Análise de Técnicas de Implementação Paralela para Treinamento de Redes Neurais em GPU*". Concluída a exposição, a candidata foi arguida pela Banca Examinadora que emitiu o seguinte parecer: "*Aprovado*". Assim sendo, deve a Universidade Federal da Paraíba expedir o respectivo diploma de Mestre em Informática na forma da lei e, para constar, eu, Alisson Vasconcelos de Brito, Coordenador do PPGI, servindo de secretário, lavrei a presente ata que vai assinada por mim e pelos membros da Banca Examinadora. João Pessoa, 31 de janeiro de 2014.

Alisson Vasconcelos de Brito

Prof. Dr. Andrei de Araújo Formiga
Orientador (PPGI-UFPB)

Prof. Dr. Thais Gaudêncio do Rego
Examinador Interno (PPGI-UFPB)

Prof. Dr. Aquiles Medeiros Filgueira Burlamaqui
Examinador Externo (UFRN)

Resumo

Com a crescente expansão do volume de dados disponíveis e a latente necessidade de transformá-los em conhecimento e informação, faz-se necessário o desenvolvimento de técnicas capazes de realizar a análise destes dados em tempo hábil e de uma maneira eficiente. Redes Neurais promovem uma análise de dados capaz de classificá-los, como também, predizem informações sobre estes. Entretanto, Redes Neurais propõem um modelo natural de computação paralela que requer técnicas de implementação com alto poder de processamento. A crescente evolução do *hardware* paralelo oferece ambientes com poder computacional cada vez mais robusto. A GPU classifica-se como *hardware* capaz de processar implementações paralelas de uma maneira eficiente e a um custo em constante redução. Sendo assim, é apresentada uma técnica de implementação paralela de Redes Neurais com processamento em GPU. Este realiza uma análise comparativa entre diferentes técnicas de implementação encontradas na literatura e a técnica proposta neste trabalho.

Palavras-chave: Redes Neurais; computação paralela; GPU.

Abstract

With the increase of data volume and the latent necessity of turn them into knowledge and information, arises the need to develop techniques able to perform the data analysis in a timely and efficient manner. Neural networks promotes an data analysis that is able to classify and predict information. However, the natural model of parallel computing proposed by neural networks, requires techniques of implementation with high processing power. The evolution of parallel hardware provides an environment with ever growing computational power. The GPU is a hardware that is able to process parallel implementations in a efficient way and at low cost. Therefore, this paper provides a technique of parallel implementation of neural networks with GPU processing and seeks to achieve an comparative analysis between different implementation techniques found in literature and the technique proposed in this paper.

Keywords: *Neural Networks, parallel computation, GPU.*

Agradecimentos

Agradeço ao DEUS Todo-Poderoso que sempre está a me guiar no seu caminho aplanado e santo, dando-me, a todo o momento, sabedoria e discernimento para prosseguir, conduzindo-me por suas veredas, levando-me em seus braços. Louvado e Engradecido seja o nome do SENHOR DEUS para todo o sempre!

Aos meus pais Jesualdo Gurgel de Almeida e Joalice Alves Ferreira Gurgel, sempre sendo minha fortaleza e meu porto seguro, as pessoas mais importantes da minha vida, que fizeram o possível e o impossível para fazer com que eu pudesse chegar até aqui, e sei que continuarão fazendo para sempre, saibam que eu os amo incondicionalmente.

Aos meus irmãos Saullo Dannylck, Sarah Johellynne e Safyra Hadassa, pelo apoio, carinho e incentivo dado por vocês durante todo esse tempo. Seria bem mais difícil sem o amor de vocês.

Aos meus sobrinhos Matheus Dannylck, Letycia Johellynne, Sofia Hadassa e Lorenna Eva, pelo simples fato de existirem, por terem vindo a esse mundo e tornado a minha vida imensamente mais feliz. Vocês ainda são pequenos, mas não imaginam a força que me dão quando simplesmente sorriem pra mim. Não há palavras para descrever o amor que sinto por vocês.

Aos meus familiares, avós, tios, tias, cunhados e primos, a todos vocês por terem acreditado em mim, mesmo que de longe, pois suas confianças e orações fizeram a diferença em minha vida.

Ao meu orientador Andrei Formiga, por todo apoio, confiança e incentivo dado a mim durante este convívio, e a todos os professores que contribuíram, dentro e fora de sala, para a formação da minha vida profissional e moral.

Aos meus queridos amigos e companheiros dessa jornada: Renata Mendonça, Elaine Duarte, Lídia Libânia, Janiclene Leite, Franciane Islânia, Paulo Medeiros, André Calisto, Berg Oliveira, Danilo Dantas, Douglas dos Santos, Luciano Fernandes, Pablo Andrey, e a todos aqui não mencionados, agradeço pela amizade conquistada, o maior ganho de todos estes anos, pelo crescimento a mim proporcionado, e por todos os momentos de alegria e de tristeza por nós vividos ao longo deste tempo.

Enfim, agradeço a todos aqueles que confiaram e contribuíram para que um dia eu pudesse chegar onde estou, saibam que esta vitória também pertence a cada um de vocês que acreditaram que tudo isto seria possível. Que Deus abençoe, guie e conduza a cada um de vocês!

Lista de Símbolos

CUDA - *Compute Unified Device Architecture*

GPU - *Graphics Processing Unit*

GPGPU - *General Purpose Graphics Processing Units*

SP - *Streaming Processors*

SM - *Streaming Multiprocessors*

Lista de Figuras

Figura 1: Neurônio biológico. 6
Figura 2. Descrição de um neurônio simples. 7
Figura 3: Perceptron Multicamadas com duas camadas ocultas. 8
Figura 4: Arquitetura típica de uma GPU. A GPU contém múltiplos *Streaming Multiprocessors* (SM), onde cada um desses é composto por múltiplos *Streaming Processors* (SP) ou *cores*. 12
Figura 5: Disposição dos elementos de um *kernel*. Um *grid* é composto por múltiplos *blocks* que compartilham a memória global entre si. Cada *block* possui múltiplas *threads* que dispõem de uma memória individual e também compartilham um memória entre as outras *threads* que pertencem ao mesmo *block*. 13
Figura 6: Organização dos *kernels* da GPU para execução de uma rede neural com 3 camadas. Cada *kernel* da GPU é responsável por computar um única camada da rede para todos os casos de entrada de um conjunto de dados em paralelo. 19
Figura 7: Organização de memória para o *kernel* que executará a propagação para frente sobre uma única camada. Os vetores de entrada e saída contém dados associados com $\boldsymbol{c}$ casos de entrada, para uma rede com $\boldsymbol{m}$ saídas e $\boldsymbol{n}$ entradas. A entrada $\boldsymbol{ijk}$ é a entrada $\boldsymbol{k}$ para o caso $\boldsymbol{j}$, e de igual modo para a saída $\boldsymbol{ojk}$. 20
Figura 8: Organização de memória para armazenamento dos valores dos pesos de cada neurônio de uma camada para todos os casos de entrada de uma rede neural. O atributo $\boldsymbol{wjic}$ corresponde ao valor do peso $\boldsymbol{i}$ do neurônio $\boldsymbol{j}$ pertencente a uma das camadas da rede neural do caso de entrada $\boldsymbol{c}$. 20
Figura 9: Alocação das *threads* em um vetor de derivadas do erro. Dado o vetor de derivadas de erro contendo o valor equivalente a cada derivada do erro de cada peso do neurônio $\boldsymbol{i}$ presente na rede neural para cada caso de entrada $\boldsymbol{c}$, cada *threads* será responsável pelo valor da mesma derivada de erro em todos os casos de entrada. 23

Figura 10: *Kernel* responsável por executar a propagação da frente por camada. 24
Figura 11: Função responsável por calcular o valor sigmóide. 24
Figura 12: *Kernel* responsável por calcular o valor delta para a camada de saída. 24
Figura 13: *kernel* responsável por calcular o valor delta para a camada h. 25
Figura 14: *kernel* responsável pelo cálculo do valor da derivada para cada peso por camada. 25
Figura 15: *kernel* responsável por atualizar os pesos da rede neural. 25
Figura 16: Comparativo entre as acelerações da implementação sequencial e da implementação T 08 sobre a implementação proposta de acordo com a quantidade de casos de entrada. 35
Figura 17: Tempo de execução na GPU e aceleração sobre a versão sequencial para o treinamento de redes neurais aumentando-se o número de casos de entrada selecionados aleatoriamente a partir do conjunto de dados *adult*. 36

Lista de Tabelas

Tabela 1.1: Definição do problema para este trabalho 3
Tabela 4.1: Resultado da pesquisa do mapeamento do estudo sistemático 14
Tabela 4.2: Trabalhos selecionados 15
Tabela 6.1: Tempo de execução do treinamento para três conjunto de dados usando versões das implementações da técnica sequencial, da técnica proposta e da técnica encontrada em T 08. 34
Tabela 6.2: Tempo de desempenho para o treinamento da rede neural com aumento do número de casos de entrada selecionados aleatoriamente a partir do conjunto de dados *adult*. A coluna aceleração descreve quantas vezes a implementação da técnica proposta é mais rápida que a técnica apresentada em T08. 35

Sumário

1. Introdução 1

1.1 Motivação 2

1.2 Definição do Problema 2

1.3 Objetivos 3

1.3.1 Objetivos Específicos 3

1.4 Metodologia 3

1.5 Publicações Relacionadas 4

1.6 Estrutura da Proposta 4

2. Redes Neurais 5

2.1 Neurônios artificiais 6

2.2 Modelos de Redes Neurais 8

2.2.1 Perceptron 8

2.2.2 Multicamadas 8

2.3 Treinamento 8

2.3.1 Propagação para Frente (*Forward Propagation*) 9

2.3.2 Retropropagação do Erro (*Backpropagation of Error*) 9

3. Computação de Propósito Geral para *Hardware* de Processamento Gráfico 11

3.1 *Graphics Processing Unit* (GPU) 11

3.2 CUDA 12

4. Trabalhos Relacionados 14

5. Redes Neurais Paralelas em GPU 18

5.1 Técnica de implementação proposta 18

5.1.1 Organização da memória 19
5.1.2 Treinamento 21
5.1.3 Implementação em CUDA 23
5.2 Técnicas de Implementações Relacionadas 26
5.2.1 T 07: Training and applying a feedforward multilayer neural network in GPU 26
5.2.2 T 08: Parallel Training of a Back-Propagation Neural Network using CUDA .. 27
5.2.3 Técnica de implementação sequencial 28
6. Experimentos e Resultados 33
7. Conclusão 37
8. Trabalhos Futuros 38
Referências Bibliográficas 39
APÊNDICE A 41
APÊNDICE B 44

1. Introdução

Atualmente, a tendência por se disponibilizar e se coletar dados está cada vez mais crescente. Estes dados advêm das mais diferenciadas fontes, contendo os mais diversos tipos de conteúdo e uma imensidão tratando-se de quantidade. A expansão acelerada da quantidade de dados desafia as ciências responsáveis por gerenciá-los, forçando-as a incrementar constantemente a capacidade de armazenamento e de processamento para os dados disponíveis.

Segundo HAN & KAMBER (2006), a necessidade de transformar esses dados em informações e conhecimento é bastante iminente nos últimos anos. Os resultados obtidos vêm a ser úteis em diversas aplicações, que vão desde a análise de mercado, para detecção de fraude e retenção de clientes, até controle da produção e exploração científica. Grandes volumes de dados, a exemplo da tendência *Big Data*[1], nos mostram a importância de analisá-los, uma vez que esta imensidão de informação tornaria-se inútil caso não houvessem maneiras de se extrair conhecimento a partir destes.

A análise de dados fornece a possibilidade de se adquirir conhecimento sobre inúmeras questões, utilizando-se de técnicas, como, por exemplo, mineração de dados, reconhecimento de padrões, aprendizado de máquina, entre outros. Estas requerem uma capacidade computacional diretamente proporcional ao volume de dados disponíveis a serem analisados.

A técnica de análise de dados proposta pela utilização de Redes Neurais, classificada como uma técnica de aprendizado de máquina, tem sido estudada como uma alternativa para promover de classificação e predição dos dados. Uma rede neural é capaz de adquirir conhecimento a partir de um conjunto de dados, e então predizer e/ou classificar o comportamento de novos dados, através do aprendizado supervisionado.

Para se realizar o processamento de uma rede neural, exige-se poder computacional elevado. Partindo do princípio de que uma rede neural possui estruturas computacionais que

[1] Prática de colecionar e processar imensos conjuntos de dados, sistemas associados e algoritmos usados para uma análise massiva desses conjuntos de dados (BEGOLI; HOREY, 2012).

sugerem um modelo natural de paralelismo, observamos que estas não obtêm tanto êxito quando são executadas a partir de uma implementação sequencial. Limitações de *hardware* fazem com que esta técnica, desde a sua idealização, seja a mais comumente utilizada.

A indústria tem encontrado avanços no desenvolvimento de *hardware* capaz de promover um ambiente para implementação do paralelismo, e a GPU (*Graphics Processing Unit*) é um exemplo deste. A princípio, a GPU era um processador dedicado exclusivamente ao processamento gráfico, porém, a sua arquitetura paralela está cada vez mais sendo direcionada para ser utilizada por programas de propósito geral.

Este trabalho propõe uma técnica de implementação paralela de Redes Neurais utilizando-se o processamento da GPU. Serão analisadas as diferentes formas de se aplicar o paralelismo a esta técnica, realizando uma comparação entre as alternativas de implementação presentes na literatura, sendo assim, servirá como um guia para futuros estudos e implementações.

1.1 Motivação

Durante anos, as barreiras criadas pelo *hardware* impossibilitaram o desenvolvimento da técnica de Redes Neurais, e direcionaram as pesquisas ao objetivo de encontrar maneiras para diminuir o volume de dados a ser processado pela rede, para que o tempo de execução também seja reduzido. Porém, com o advento de tecnologias que proporcionam um ambiente de programação paralela tornou-se possível estudar técnicas que efetivamente implementam redes neurais paralelamente.

Portanto, diante do recente advento do *hardware* de processamento paralelo, sendo este uma solução que cada vez mais tem o seu custo reduzido e seu poder de processamento aumentado, e a necessidade de técnicas que promovam eficientemente a extração de informações e conhecimentos a partir de grandes volumes de dados, define-se a importância de se realizar esta pesquisa e avançar com os estudos na implementação paralela de uma Rede Neural para processamento em GPU.

1.2 Definição do Problema

A definição do problema a ser estudado neste trabalho está representada na Tabela 1.1, utilizando-se do modelo *Goal, Question, Metric* (CALDIERA; ROMBACH, 1994), podemos visualizar da seguinte maneira:

Tabela 1.1: Definição do problema para este trabalho

Analisar técnicas de implementação paralela de Redes Neurais **com a intenção de** encontrar soluções de alto poder de processamento **com respeito à** análise de grandes volumes de dados **do ponto de vista da** comunidade acadêmica e da indústria **no contexto da** gestão estratégica da informação.

1.3 Objetivos

O objetivo geral deste trabalho é propor uma técnica de implementação paralela de Redes Neurais para GPU.

1.3.1 Objetivos Específicos

A fim de se alcançar o objetivo geral deste trabalho, os seguintes objetivos específicos são propostos:

1. Criar uma implementação paralela de Redes Neurais para GPU;
2. Analisar técnicas de implementação semelhantes a proposta;
3. Realizar comparações entre as técnicas de implementação paralela encontradas na literatura e a implementação proposta neste trabalho, analisando suas formas de implementação, estratégias e desempenho de execução;

1.4 Metodologia

Serão realizadas atividades que proponham uma pesquisa exploratória sobre o tema implementação paralela de Redes Neurais. Será feita uma revisão na literatura com o objetivo de se encontrar trabalhos que tenham propostas semelhantes a esta.

Será apresentada a técnica desenvolvida para este trabalho que foi utilizada para criar a implementação paralela de Redes Neurais para processamento em GPU.

Experimentos entre as técnicas encontradas serão realizados, a fim de se obter resultados que demostrem os seus desempenhos de execução para assim realizar uma análise comparativa entre estas e a técnica proposta neste trabalho. Serão identificadas as vantagens e desvantagens de cada técnica, como também, situações nas quais estas se aplicam melhor.

Conjuntos de dados, preferencialmente, que possuam grandes volumes, serão analisados com o objetivo de utilizar ao máximo o poder de processamento proposto pelas técnicas.

1.5 Publicações Relacionadas

Esta pesquisa tornou possível a publicação de um artigo no *2nd Brazilian Conference on Intelligent Systems* (BRACIS-13), intitulado "*Parallel Implementation of Feedforward Neural Networks on GPUs*". O artigo tem como objetivo explanar sobre a implementação paralela de Redes Neurais desenvolvida durante a realização deste trabalho, relatando sobre as técnicas utilizadas para paralelizar a implementação e demostrando os resultados do experimento realizado comparando-se a implementação proposta com uma implementação sequencial. Os resultados encontrados mostram superioridade no desempenho de execução do algoritmo proposto, e estão de acordo com o objetivo da proposta.

1.6 Estrutura da Proposta

Esta proposta está dividida da seguinte maneira:

- Capítulo 2: Apresenta a fundamentação teórica sobre o tema Redes Neurais, abordando alguns assuntos necessários para o desenvolvimento deste trabalho;
- Capítulo 3: Explana sobre computação de propósito geral para *hardware* de processamento gráfico;
- Capítulo 4: Apresenta os trabalhos relacionados a esta pesquisa;
- Capítulo 5: Expõe a técnica de implementação paralela para Redes Neurais em GPU proposta nesta pesquisa e outras técnicas de implementação encontradas na literatura;
- Capítulo 6: Apresenta os experimentos e resultados; por fim, o
- Capítulo 7: Apresenta a conclusão deste trabalho.

2. Redes Neurais

O cérebro humano tem aproximadamente cem bilhões de neurônios. Este surpreendente número de células, que é utilizado como referência padrão na literatura que envolve a neurociência, está em constante processo de mudança e evolução. O cérebro humano tende a crescer dada a evolução da espécie, contribuindo assim para o surgimento de novas células (LENT et al., 2012).

Levando-se em consideração um número tão grande de neurônios, percebe-se que é necessária a utilização de uma estrutura altamente complexa, robusta e eficiente para que estas células possam trabalhar, seja armazenando, atualizando, compartilhando e/ou executando outros procedimentos inerentes às suas atividades. No corpo humano, esta estrutura chama-se sistema nervoso, tendo como principal órgão o cérebro. Segundo (KOCH, 1999), o cérebro "é muitas vezes descrito como o sistema mais complexo do universo". É no cérebro onde ocorre toda a comunicação dos neurônios. Deve-se levar em consideração também que a comunicação entre essas células é feita através de outros bilhões de linhas de comunicação. Ainda segundo KOCH (1999), "Estamos começando a entender os códigos utilizados pelos impulsos dos neurônios para transmitir informações do ambiente da periferia para estruturas mais profundas do cérebro.".

Um neurônio (Figura 1) é uma célula composta por um corpo celular, também chamado de soma, que é o centro de metabolismo da célula; pelo axônio, responsável pela transmissão de mensagens aos outros neurônios ou a um músculo; e pelos dendritos, responsável pelo recebimento de mensagens de outros neurônios. O fluxo de mensagens entre neurônios ocorre a partir do axônio de um neurônio para um dendrito de outro neurônio, a este acontecimento é dado o nome de sinapse.

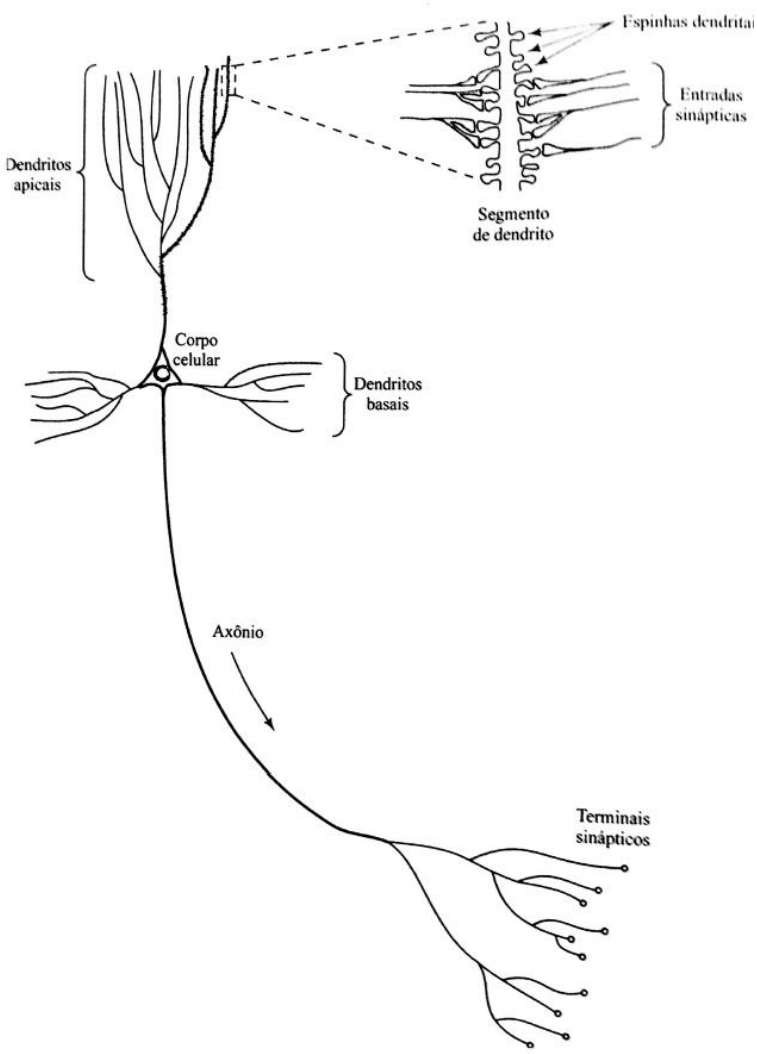


Figura 1: Neurônio biológico.
Fonte: SIMON (2001)

A capacidade de mudar a quantidade de carga necessária para gerar uma sinapse é o fator que torna possível a aprendizagem de um neurônio, isto possibilita a realização de atividades como reconhecer objetos, aprender idiomas, memorizar informações, entre outras. (KANDEL; SCHWARTZ; JESSELL, 2000).

2.1 Neurônios artificiais

O estudo do funcionamento de um neurônio possibilitou que a forma de aprendizagem realizada fosse aplicada em um modelo. Os pesquisadores MCCULLOCH & PITTS (1943) propuseram um exemplo da modelagem de um neurônio: um conjunto de pesos de entrada: $w_1, w_2, \dots, w_i$, que corresponde às sinapses; um somatório: que soma os sinais de entrada associados aos pesos e ao viés, equivale à colina axônica; e uma função de ativação: que decide se gera ou se não gera o impulso para as entradas recebidas (MARSLAND, 2009).

Sendo assim, um neurônio (artificial) é um elemento que calcula uma saída y a partir de um conjunto de entradas $x_1, x_2, x_3, \dots, x_n$. A saída é calculada a partir de uma função de ativação $a(x)$ que realiza uma combinação linear da série de entradas.

Uma rede neural é composta por um ou mais neurônios interligados entre si, de modo que a saída de um neurônio corresponde à entrada de outro neurônio. A seguir, a representação do neurônio modelado por MCCULLOCH & PITTS, que também descreve um exemplo mais simples de uma rede neural:

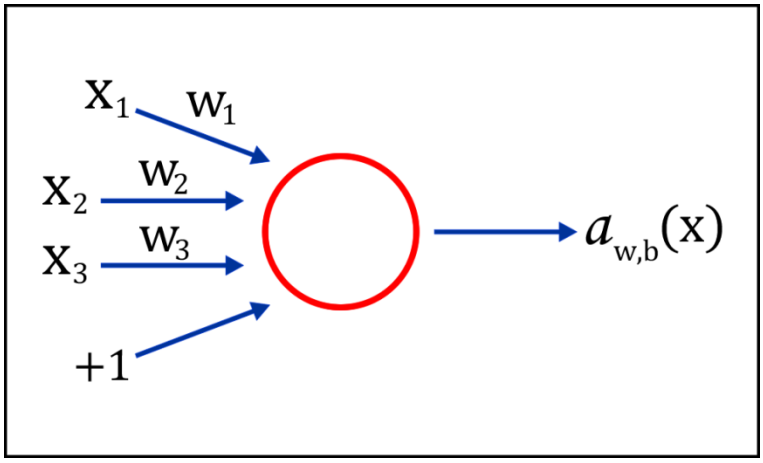


Figura 2. Descrição de um neurônio simples.

Este neurônio computa o somatório conjunto de entradas x_1, x_2, x_3, associadas ao conjunto de pesos w_1, w_2, w_3 e ao viés[2] $+1$ e gera a saída correspondente, sendo n o número correspondente à quantidade de entrada e b o valor atribuído ao viés, para este exemplo, temos como função $a = f(\sum_{i=1}^{n} w_i x_i + b)$.

A função limiar é comumente utilizada como função de ativação, dado que, esta modela a característica de um resultado binário. A saída y da função será:

$$f(a) = \begin{cases} 1, se\ a > 0 \\ 0, se\ a \leq 0 \end{cases}$$

A função sigmóide fornece uma maior precisão no resultado de acordo com HAYKIN (2001), tendo em vista que esta promove um balanceamento adequado entre comportamento linear e não-linear. A sua representação é dada pela fórmula $f(a) = \frac{1}{1+e^{-a}}$, onde os resultados assumem valores que variam entre 0 e 1.

[2] Representa um estímulo inicial à rede neural

2.2 Modelos de Redes Neurais

2.2.1 Perceptron

O modelo Perceptron é constituído por um conjunto de neurônios de MCCULLOCH & PITTS, possui apenas uma camada de m entradas $\{m_1, m_2, \dots, m_n\}$ interligadas a uma outra camada com apenas um neurônio através dos pesos $\mathrm{w} = [\mathrm{w}_1, \mathrm{w}_2, \mathrm{w}_3, \dots, \mathrm{w}_n]$. A saída é gerada pela função de ativação presente no neurônio da segunda camada.

2.2.2 Multicamadas

O modelo Perceptron Multicamadas possui duas ou mais camadas de neurônios, partindo do princípio de que este terá, no mínimo uma camada para a entrada e uma outra camada para a saída. A rede também poderá conter uma ou mais camadas com neurônios ocultos, que não pertencem às camadas de entrada ou de saída. Desta forma, cada camada l_c terá n neurônios $\{n_1, n_2, \dots, n_j\}$ que estarão interligados aos pesos advindos da camada anterior (l_{c-1}) que possui m neurônios $\{m_1, m_2, \dots, m_i\}$. Os pesos w estão organizados na seguinte matriz $\begin{bmatrix} w_{11} & \cdots & w_{1j} \\ \vdots & \ddots & \vdots \\ w_{i1} & \cdots & w_{ij} \end{bmatrix}$. Veja a Figura 3.

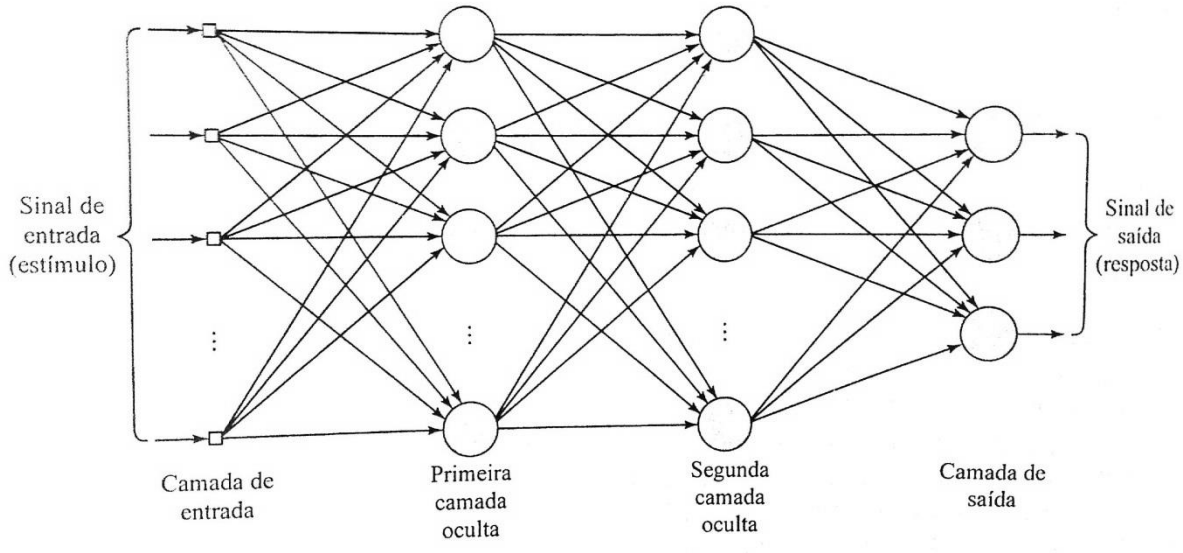


Figura 3: Perceptron Multicamadas com duas camadas ocultas.
Fonte: SIMON (2001)

2.3 Treinamento

O aprendizado da rede é adquirido através de treinamento. Os pesos sinápticos vão sendo alterados ao decorrer da execução do algoritmo; este recebe como entrada um conjunto

de dados com pares de um vetor de entrada e um de saída, o valor da comparação entre a saída desejada e a saída encontrada juntamente com o termo taxa de aprendizagem, serão os responsáveis por fazer as alterações nos valores dos pesos, fazendo com que a rede aprenda de acordo com os dados recebidos e, ao final desta etapa, possa dar um resultado coerente para novas entradas, não pertencentes ao conjunto de treinamento.

2.3.1 Propagação para Frente (*Forward Propagation*)

Para a inicialização deste algorítmo, devem ser atribuídos aos pesos valores aleatórios próximos a zero (positivos e negativos). Para a etapa específica de treinamento, serão executadas iterações para cada vetor de entrada, onde será computada a função de ativação para cada neurônio da rede; ao final de cada iteração, os valores dos pesos são atualizados de acordo com a seguinte expressão: $\Delta w_{kj}(n) = \eta e_k(n) \cdot a_j(n)$, onde η é uma constante positiva que determina o valor da taxa de aprendizagem, que decide o quão rápido será o aprendizado da rede; e_k referencia o sinal de erro[3] dado pela fórmula $e_k(n) = d_k(n) - y_k(n)$, sendo d_k é o resultado desejado da função de ativação do neurônio k e y_k a saída encontrada para o mesmo; e a_j representa a função de ativação para o neurônio k.

2.3.2 Retropropagação do Erro (*Backpropagation of Error*)

Em um primeiro momento segue-se o modelo da Propagação para Frente, até o ponto em que é calculado o valor de saída para todos os neurônios da rede. Nesta etapa, os valores dos pesos permanecem inalterados.

Em seguida, inicia-se o processo de retropropagação em si, computa-se o valor das derivadas dos erros em relação aos pesos para cada neurônio da rede, partindo da última camada (a camada de saída) em direção à primeira camada (a camada de entrada), o objetivo deste cálculo é identificar o peso que seria o maior responsável pelo erro da série.

A computação da derivada de erros tem o objetivo de fornecer dados para a atualização dos pesos e reduzir o valor do erro. É realizada a partir da obtenção do valor delta de cada neurônio, que é computado da seguinte forma: na camada de saída, calcula-se a diferença entre a saída desejada e a saída obtida, e multiplica-se seu valor invertido com uma função de ativação para sua saída, utiliza-se a expressão $\delta_i = -(d_i - y_i) \cdot a'(y_i)$ para representar a mesma, onde cada neurônio i, possui uma saída desejada d, e, para as camadas anteriores, computa-se o somatório da multiplicação entre os pesos que fazem ligação de um

[3] O objetivo do sinal de erro, representador por e_k, é promover um ajuste corretivo aos pesos sinápticos do neurônio k.

neurônio com a camada posterior, o valor delta deste neurônio e o resultado da sua função de ativação, é dada pela expressão $\delta_i = \sum_k w_{ki} \cdot \delta_k \cdot a'(y_i)$.

No momento em que cada neurônio possui seu próprio valor delta, é realizada a computação da derivada, esta é representada pela seguinte expressão $\frac{\partial E_P}{\partial w_{ik}} = \delta_i y_k$, ou seja, a derivada do peso w_i interligado ao neurônio k do caso de teste P é o resultado da multiplicação entre o seu valor delta e o valor da saída do neurônio k.

A atualização dos pesos se dá pela computação da seguinte expressão

$$w'_{ik} = w_{ik} - \eta \frac{\partial E}{\partial w_{ik}}$$

onde o novo peso é obtido subtraindo-se o valor da multiplicação entre o somatório da sua derivada entre todos os casos de entrada e a taxa de aprendizado da rede neural.

Este algoritmo tende a obter maior precisão na atualização dos valores, pois o erro é encontrado e corrigido individualmente a cada neurônio, sendo assim, o aprendizado da rede torna-se mais eficiente, dada a criteriosa atualização dos pesos.

3. Computação de Propósito Geral para *Hardware* de Processamento Gráfico

Com a recente evolução do *hardware* de processamento gráfico, tornou-se possível executar nestes, tarefas direcionadas não apenas à computação gráfica, como também, para a computação de propósito geral.

A princípio, até a década de 1980, o *hardware* gráfico realizava apenas o processamento de tarefas bastante simples, como desenhar linhas, arcos e caracteres. Na década seguinte, a evolução do *hardware* proporcionou a inclusão dos dispositivos chamados *shaders*, que possuíam funções gráficas simples embutidas. Na década de 2000, acrescentaram-se recursos para que se permitisse executar instruções diretamente nos *shaders*. Nos recentes anos, os *shaders* passaram a suportar a execução de instruções de mais alto nível, evoluindo o conceito do termo para os chamados *kernels*. Esta evolução resultou na introdução do termo GPGPU (Unidades de Processamento Gráfico de Propósito Geral).

3.1 *Graphics Processing Unit* (GPU)

A GPU trata-se de um *hardware* com imenso poder computacional de processamento paralelo. Sua arquitetura *multicore* e *multithread* foi elaborada com o objetivo de realizar tarefas envolvidas na renderização gráfica para computadores pessoais. Com o tempo, foram sendo realizadas adaptações no *hardware* para que se processasse implementações de propósito geral.

As GPUs possuem vários núcleos de processamento simples, os chamados *Streaming Processors* (SP), que chegam a totalizar centenas. Os SPs são agrupados por estruturas chamadas *Streaming Multiprocessors* (SM). Para cada SM existe uma memória que é compartilhada por todas os seus SPs. A Figura 4 ilustra uma arquitetura de uma GPU:

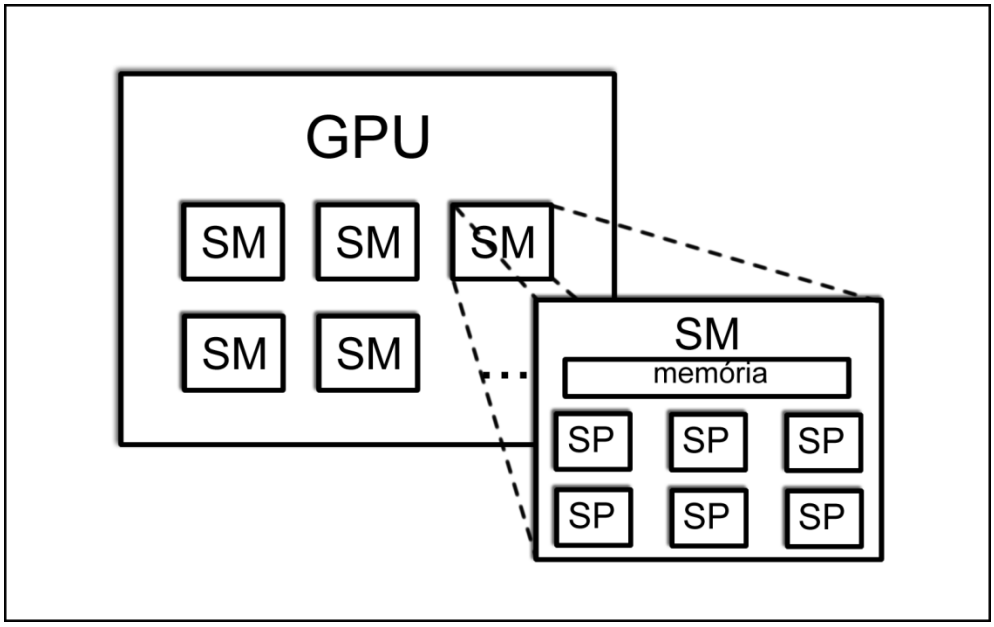


Figura 4: Arquitetura típica de uma GPU. A GPU contém múltiplos *Streaming Multiprocessors* (SM), onde cada um desses é composto por múltiplos *Streaming Processors* (SP) ou *cores*.

3.2 CUDA

CUDA (*Compute Unified Device Architecture*), é um modelo de programação paralela e um ambiente de *software* projetado para desenvolver aplicativos que tratam de maneira transparente a utilização do paralelismo das GPUs. A curva de aprendizado em CUDA é bastante suave para programadores familiarizados com as linguagens padrão, como, por exemplo, a linguagem C, o que torna seu desenvolvimento ainda mais confortável (NVIDIA, 2008).

O ambiente CUDA torna capaz a utilização da arquitetura da GPU para implementar soluções em programação paralela. Sendo a GPU composta de vários processadores, a CUDA proporciona um ambiente que executa várias *threads* em paralelo, dividindo-as entre seus processadores.

A computação paralela no ambiente CUDA ocorre da seguinte maneira: as partes de um código que podem ser executas em paralelo são traduzidas para CUDA e executadas em GPU. Esses trechos são reescritos na forma de *kernels* paralelos. Um *kernel* comanda a execução de um conjunto de *threads* na GPU, onde estes são organizados em *grids* que são compostos pelos chamados *blocks*, que são formados por um conjunto de *threads*. Uma *grid* é um conjunto *blocks* que executa independentemente, enquanto que *block* é um conjunto de *threads* que pode cooperar por meio de sincronização do tipo barreira e acesso compartilhado

a um espaço de memória exclusivo de cada bloco de *thread*. A Figura 5 apresenta a organização descrita anteriormente para o *kernel*.

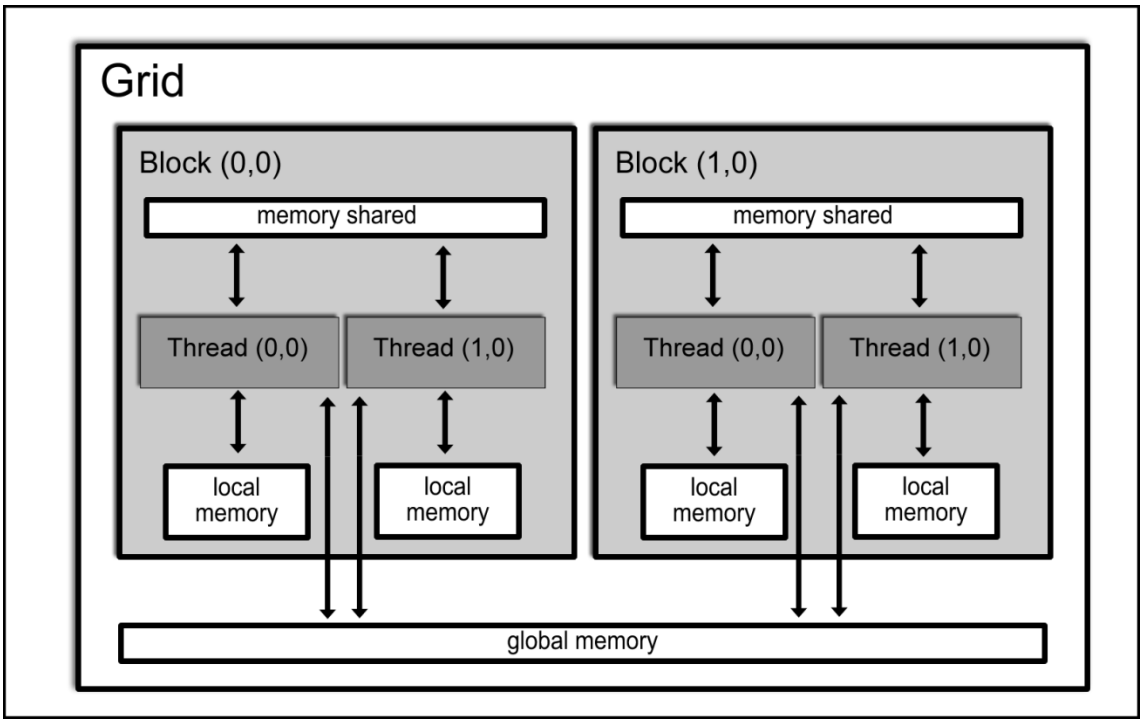


Figura 5: Disposição dos elementos de um *kernel*. Um *grid* é composto por múltiplos *blocks* que compartilham a memória global entre si. Cada *block* possui múltiplas *threads* que dispõem de uma memória individual e também compartilham um memória entre as outras *threads* que pertencem ao mesmo *block*.

Para a execução de um *kernel*, é realizada uma chamada a este a partir de uma declaração global que indica o número de *threads* e de *blocks* para cada grid. Cada processo vai ser executado em uma *thread*, onde cada uma desta possui um número de identificação.

4. Trabalhos Relacionados

Implementações de redes neurais para GPU vêm sendo pesquisadas e desenvolvidas desde que a GPU tornou-se alvo da computação de propósito geral. Sendo assim, esta seção aborda uma análise sobre os trabalhos relacionados que foram encontrados na literatura que estão correlacionados ao tema proposta no presente trabalho.

Para a escolha dos trabalhos, foi realizada uma pesquisa baseada no padrão de mapeamento do estudo sistemático. A pesquisa reuniu trabalhos relacionados à temática Redes Neurais paralelas em GPU, e seguiu os critérios expostos no protocolo que encontra-se no Anexo I deste trabalho.

Os resultados obtidos na pesquisa podem ser observados na Tabela 4.1: Resultado da pesquisa do mapeamento do estudo sistemático, esta descreve a quantidade de arquivos que foram encontrados de acordo com a *string* de busca formatada em cada base de dados, como também a quantidade de arquivos que foram selecionados dentre estes.

Tabela 4.1: Resultado da pesquisa do mapeamento do estudo sistemático

Base de Dados	***String* de Busca Formatada**	**Quantidade de trabalhos encontrados**	**Quantidade de trabalhos selecionados**
IEEE *Xplore*	(("*Document Title":Neural* OR "*Document Title":Neuronal*) AND ("*Document Title":Network* OR "*Document Title":Networks*)) AND ("*Document Title"*:GPU OR "*Document Title*":CUDA OR "*Document Title":Graphic Processing Unit*)	26	3
Google Scholar	((notítulo:*Neural* OR notítulo:*Neuronal*) AND (notítulo:*Network* OR notítulo:*Networks*)) AND (notítulo:GPU OR notítulo:CUDA OR notítulo:*Graphic Processing Unit*)	81	2

Science Direct	((*Title(Neural*) OR *Title(Neuronal))* AND (*Title(Network)* OR *Title(Networks*))) AND (*Title(*GPU) OR *Title*(CUDA) OR *Title(Graphic Processing Unit))*	3	3

A partir desta busca, foi selecionado um total de 8 (oito) trabalhos de acordo com o protocolo do mapeamento do estudo sistemático. A Tabela 4.2: Trabalhos selecionados descreve os títulos e as referências dos trabalhos selecionados

Tabela 4.2: Trabalhos selecionados

Nº	Título do Artigo	Referência
T 01	*GPU implementation of neural networks*	(OH; JUNG, 2004)
T 02	*Parallelization of cellular neural networks on GPU*	(HO; LAM; LEUNG, 2008)
T 03	*A multi-GPU algorithm for large-scale neuronal networks*	(CAMARGO; ROZANTE; SONG, 2011)
T 04	*GPU-based simulation of cellular neural networks for image processing*	(DOLAN; DESOUZA, 2009)
T 05	*Parallel implementation of neural networks training on graphic processing unit*	(LIU et al., 2012)
T 06	*Multicore and GPU Parallelization of Neural Networks for Face Recognition*	(AHMAD et al., 2013)
T 07	*Training and applying a feedforward multilayer neural network in GPU*	(RAIZER; IDAGAWA, 2009)
T 08	*Parallel Training of a Back-Propagation Neural Network Using CUDA*	(SIERRA-CANTO; MADERA-RAMIREZ; UC-CETINA, 2010)

Os trabalhos **T 01** e **T 02** propõem a utilização dos componentes da GPU para processamento de redes neurais, sendo que estes possuem um nível elevado de dificuldade para desenvolvimento, devido a não existência de um ambiente voltado para este tipo de programação.

Com o advento do ambiente CUDA, novas pesquisas foram realizadas e novos trabalhos foram desenvolvidos com o objetivo de paralelizar diferentes modelos de redes neurais em GPU. O artigo **T 03** apresenta uma implementação em CUDA para simulação em larga escala de uma rede neural composta por neurônios biologicamente realistas Hodgkin–Huxley em um ambiente de múltiplas GPUs. O algoritmo é dividido em duas partes: na primeira, a cada neurônio da rede é atribuído uma *thread* simples para cálculo de sua equação; na segunda, utiliza-se do processamento em CPU para comunicação dos neurônios, que podem estar localizados em diferentes GPUs. A computação oferecida pela GPU foi utilizada

basicamente para processamento de cálculo de vetores e matrizes, não tendo seu ambiente aproveitado de uma melhor forma.

No artigo **T 04**, os autores apresentam uma solução em GPU para simular redes neurais celulares utilizando da plataforma CUDA. No modelo de rede neural utilizado, que é semelhante ao modelo de redes neurais artificiais, cada elemento, chamado célula, realiza o processamento de sua informação e envia o resultado aos seus vizinhos. Este modelo sugere naturalmente uma solução paralela, onde o processamento de cada célula é realizado simultaneamente. A simulação, que tem o objetivo de realizar processamento de imagem, foi desenvolvida para três plataformas: CPU *single-core*, CPU *multi-core* e GPU. Embora a implementação em CUDA para a GPU tenha algumas limitações para trabalhar com OpenCV, esta mostrou-se mais rápida do que as implementações sequenciais.

O trabalho **T 05** propõe uma solução para treinamento de um modelo acústico utilizando o modelo de redes neurais profundas. O trabalho apresenta uma implementação paralela em ambiente CUDA para o algoritmo de propagação para trás para o treinamento da rede neural. Uma implementação sequencial foi desenvolvida para trabalhar em conjunto com a paralela, onde a execução em GPU ficou responsável pela computação mais complexa, envolvendo somatórios e multiplicação de matrizes, e a execução em CPU responsável por funções mais simples como gerenciamento de memória. Técnicas de redução em GPU foram aplicadas para melhores resultados. Foi realizado um comparativo da execução paralela com uma implementação sequencial que utiliza a Intel® MKL, onde a solução paralela mostrou-se até 26 vezes mais rápida.

A pesquisa proposta em **T 06** apresenta uma implementação de uma rede neural artificial em ambiente CUDA para a técnica de reconhecimento de face. Para treinamento da rede, utiliza-se do algoritmo de propagação para trás, onde, na fase de propagação para a frente, cada *thread* foi alocada para realizar o cálculo do resultado para cada neurônio, e, na fase de ajuste dos pesos, foi alocada uma *thread* para cada peso. Sendo assim, o processamento para os neurônios de cada camada é feito em paralelo, da mesma forma, para os pesos de uma camada. Observando que o treinamento em épocas é realizado sequencialmente para cada caso do conjunto de dados de entrada, o que custará um tempo não tão favorável quando o conjunto possuir um grande volume de dados.

Os trabalhos **T 08** e **T 09** apresentam técnicas de implementações em GPU utilizando o ambiente CUDA para o treinamento de uma rede neural artificial com o algoritmo de retropropagação. A rede neural utilizada em ambos é limitada a três camadas: a de entrada,

uma camada escondida e a camada de saída. Para o primeiro trabalho, o recurso matrixMul, proveniente da SDK CUDA, é utilizado para computação de vetores e matrizes. Enquanto que, no segundo trabalho citado, esta computação é realizada pela biblioteca CUBLAS. O treinamento ocorreu em épocas, onde, para cada época, os casos de teste foram executados sequencialmente. Para avaliar o desempenho, variou-se a quantidade de neurônios da única camada escondida.

5. Redes Neurais Paralelas em GPU

Este capítulo descreve a técnica para implementação de Redes Neurais paralelas em GPU proposta para este trabalho; como também, explana sobre outras técnicas encontradas na literatura que serviram para a realização de uma análise comparativa entre estas.

5.1 Técnica de implementação proposta

Temos em uma rede neural um modelo de paralelização natural, onde os seus neurônios podem ser associados diretamente às *threads* da GPU, sendo que, o passo paralelo seja a computação da saída de um neurônio por uma *thread* e esta execução ocorra em todos os neurônios da rede simultaneamente. Porém, sabemos que o cálculo da saída de um neurônio depende dos resultados da camada anterior, tratando-se de propagação para frente, e, no caso da retropropagação, os resultados dependem das saídas da camada posterior, logo, este modelo não seria funcional.

Analisando as etapas do processo de treinamento de uma rede neural com retropropagação, podemos observar que são necessários vários casos de entrada para obter-se um bom resultado de treinamento. Sabemos que cada camada de um caso de entrada possui os mesmos requisitos para ser computada, comparando-se as mesmas camadas dos outros casos de entrada. Sendo assim, a paralelização pode ocorrer ao mapearmos um *kernel* para a computação de cada camada da rede, tendo a mesma camada de todos os casos de entrada atribuída para um bloco de *threads* diferente, e o conjunto de neurônios de cada camada teria a computação individual de cada neurônio por cada *thread* do bloco em que sua camada se encontra. Desta forma, a execução seria controlada para que, a cada passo, apenas uma camada de cada caso de entrada seja computada. A Figura 6 ilustra esta organização:

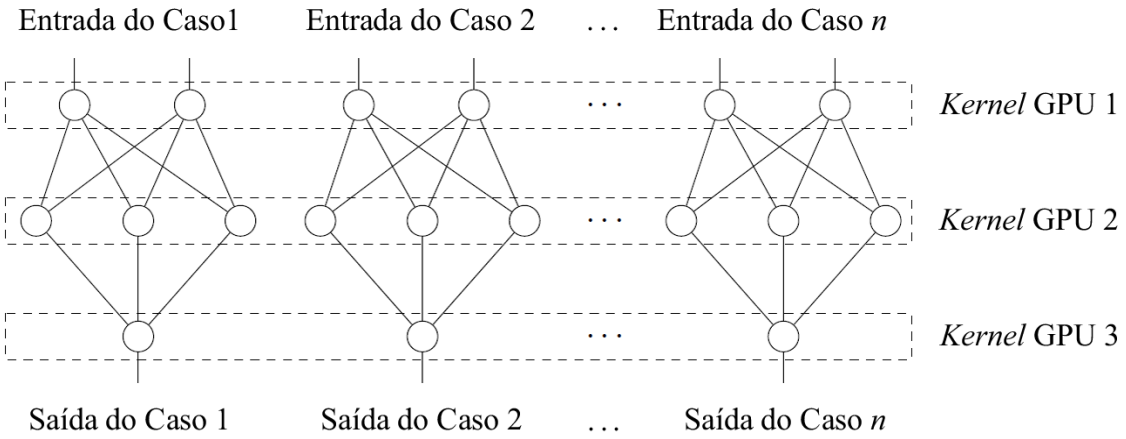


Figura 6: Organização dos *kernels* da GPU para execução de uma rede neural com 3 camadas. Cada *kernel* da GPU é responsável por computar um única camada da rede para todos os casos de entrada de um conjunto de dados em paralelo.

5.1.1 Organização da memória

Um dos grandes desafios ao se programar em um modelo paralelo é escolher uma organização de memória capaz de extrair o máximo de paralelismo possível. Esta etapa requer bastante análise para que se obtenha tal modelo eficaz.

Tendo em vista o modelo de paralelização proposto, observamos necessária a criação de vetores lineares para o armazenamento dos valores necessários à computação dos dados em uma rede neural. A princípio, temos duas estruturas que armazenam todos os valores de entrada e de saída de uma camada para todos os casos de entrada. Isto se faz necessário devido ao paralelismo ocorrer por camada em todos os casos de entrada, sendo que, a cada passo simultâneo, só são necessários os dados referentes à camada que está sendo computada atualmente.

O tamanho do vetor de entrada para a computação da primeira camada escondida corresponde à quantidade de entradas da rede neural; os valores subsequentes possuem o mesmo tamanho do vetor de saída da camada anterior a este. Os vetores de saída terão o mesmo tamanho da quantidade de neurônios presentes na camada correspondente a este, veja a Figura 7.

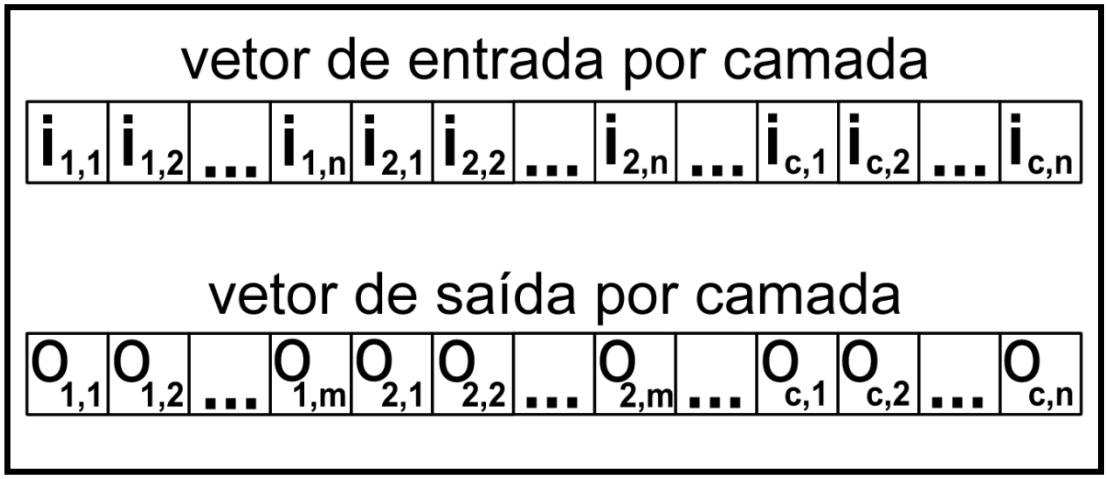


Figura 7: Organização de memória para o *kernel* que executará a propagação para frente sobre uma única camada. Os vetores de entrada e saída contém dados associados com c casos de entrada, para uma rede com m saídas e n entradas. A entrada i_{jk} é a entrada k para o caso j, e de igual modo para a saída o_{jk}.

Os valores dos pesos também se fazem necessário ser armazenados em vetores lineares para a realização da computação da saída de um neurônio. A estrutura corresponde a um vetor que armazena todos os pesos referentes a cada neurônio de uma camada para todos os casos de entrada. A representação gráfica do referido vetor é dada na Figura a seguir:

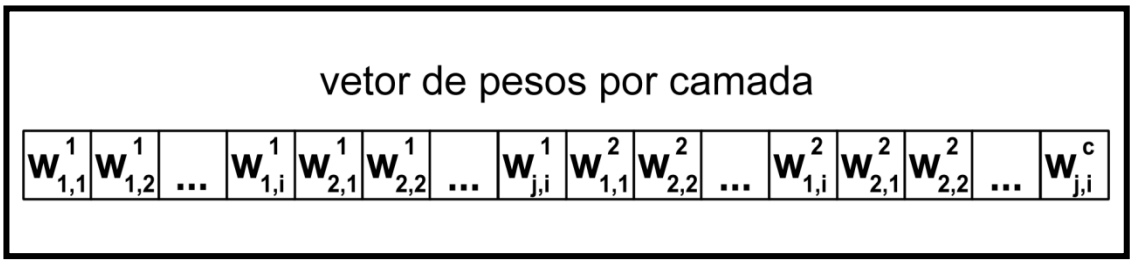


Figura 8: Organização de memória para armazenamento dos valores dos pesos de cada neurônio de uma camada para todos os casos de entrada de uma rede neural. O atributo $w_{ji}{}^{c}$ corresponde ao valor do peso i do neurônio j pertencente a uma das camadas da rede neural do caso de entrada c.

O armazenamento dos valores delta necessários na retropropagação, são armazenados do mesmo modo dos valores de saída, em um vetor linear de tamanho correspondente ao número de neurônios presente na camada repetidas vezes a quantidade de casos de entrada da rede neural. Entretanto, os valores das derivadas dos pesos são armazenados em vetores lineares semelhantes ao vetor de pesos, onde possui o valor referido da derivada de cada peso

de um neurônio em uma camada repetidas vezes a quantidade de casos de entrada da rede neural.

5.1.2 Treinamento

No processamento paralelo também optou-se por utilizar o treinamento por lotes, este método em si é executado em processamento sequencial, porém, é a partir deste que são chamados os *kernels* para a execução paralela. Para as suas chamadas recorrentes, recebem-se como parâmetros: a rede neural a ser treinada, o conjunto de dados, a quantidade de épocas e a taxa de aprendizado desejada. Assim, para cada época, realizam-se os seguintes passos:

5.1.2.1 Propagação para frente

Para cada camada da rede neural, é chamada a função que computa a saída de todos os neurônios da referida camada simultaneamente para todos os casos de entrada. Este último método é propriamente executado no dispositivo de programação paralela. Os parâmetros recebidos são: o vetor de pesos, um índice que representa onde encontra-se o valor do primeiro peso da camada no vetor de pesos, a quantidade de pesos por neurônio, e o vetor de valores de entrada. Ao tamanho dos blocos de *thread*s é atribuído o valor correspondente à quantidade de casos de entrada da rede neural, e à quantidade de *thread*s por bloco é atribuído o valor correspondente à quantidade de neurônios presentes na referida camada. Ao índice que controla o vetor de pesos é atribuído o valor do índice atual somado à multiplicação entre o índice das *thread*s e a quantidade de pesos por neurônio. A princípio, o valor do somatório de cada neurônio inicia-se com o valor do viés e em seguida realiza-se o somatório propriamente dito entre a multiplicação dos valores de entrada e o valor do peso correspondente. Em seguida, é calculado com a função de ativação desejada sobre o resultado do somatório o resultado que corresponde à saída do neurônio.

5.1.2.2 Computação do valor delta

A computação do valor delta de cada neurônio de cada camada de saída da rede neural para todos os casos de entrada é realizada ao mesmo modo do método citado no item 5.1.2.1, a execução deste é paralela. Desta forma, compara-se o vetor de saídas da referida camada com o vetor de saídas desejadas da rede neural simultaneamente para todos os casos de entrada da rede neural.

A computação dos valores delta referentes aos demais neurônios da rede também possui uma função de execução paralela. Este método é chamado para cada camada da rede,

onde são alocados blocos de *thread*s na quantidade correspondente ao número de casos de entrada, e o número de *thread*s por bloco corresponde a quantidade de neurônios presentes na referida camada. O índice da *thread* corresponde a referência do valor delta no vetor correspondente, o índice do vetor de saída corresponde ao índice de blocos multiplicado pelo número de neurônios da camada posterior, e o índice que se refere ao vetor de pesos corresponde aos valores dos pesos da camada seguinte incrementados em 1 para corresponder ao valor bias. Sendo assim, a iteração sobre o número de neurônios da camada seguinte permite a computação simultânea do resultado do delta de acordo com a função proposta na seção 2.3.2 deste trabalho.

5.1.2.3 Computação do valor da derivada do erro

A computação do valor da derivada do erro, para cada peso de cada neurônio presente na rede neural para todos os casos de entrada, também possui execução paralela, porém, possui uma diferenciação na organização dos blocos na GPU. Na iteração sobre a camada de saída até a primeira camada escondida é chamado o método para realizar este cálculo, sendo que, o tamanho do bloco de *thread*s permanece com o mesmo valor das execuções anteriores, com o valor correspondente ao número de casos de entrada. Porém, o número de *thread*s por bloco terá quantidade definida a partir da multiplicação entre o número de neurônios da camada recorrente e o número de pesos por neurônio. Sendo assim, torna-se possível a realização do cálculo da derivada do erro simultaneamente para cada peso individualmente, sendo que cada *thread* é responsável por fazer o cálculo, que consiste na multiplicação entre o valor do delta pertencente ao peso em questão e o valor da saída do neurônio que este peso está associado.

5.1.2.4 Atualização dos pesos

O método responsável pela atualização dos pesos também possui execução paralela. A organização *kernel* possui apenas 1 (um) bloco de *thread*s e a quantidade de *thread*s por bloco é equivalente ao número total de pesos contidos na rede. Sendo assim, cada *thread* é responsável por combinar o valor da derivada do erro correspondente ao peso em questão entre todos os casos de entrada (Figura 9).

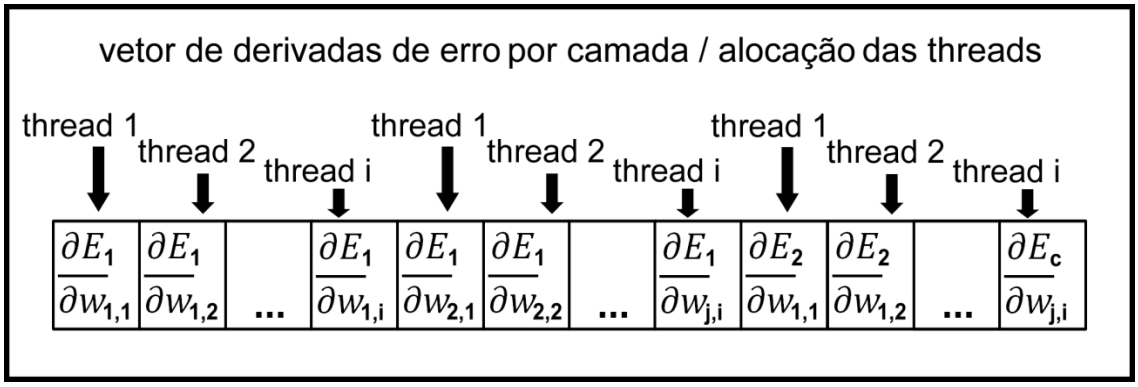


Figura 9: Alocação das *threads* em um vetor de derivadas do erro. Dado o vetor de derivadas de erro contendo o valor equivalente a cada derivada do erro de cada peso do neurônio *i* presente na rede neural para cada caso de entrada *c*, cada *threads* será responsável pelo valor da mesma derivada de erro em todos os casos de entrada.

5.1.3 Implementação em CUDA

O ambiente CUDA foi escolhido para o desenvolvimento da implementação proposta porque este oferece recursos para programação paralela em alto nível, tornando abstrato o desenvolvimento para o *hardware* de processamento gráfico.

Foram desenvolvidos *kernels* para as funções que anteriormente foram descritas como de execução paralela. A etapa que trata da iteração no treinamento em épocas é executada em CPU, porém realiza chamadas aos respectivos *kernels* para realizarem a computação necessária dos dados.

A princípio, a rede neural é montada com as camadas necessárias e as suas respectivas quantidades de neurônios, e os dados dos conjuntos de treinamentos são atribuídos aos valores do vetor de entrada. Em seguida, inicia-se a iteração das épocas, e então o *kernel* responsável por realizar a etapa de propagação para frente (Figura 10**Erro! Fonte de referência não encontrada.**) é chamado em uma iteração para cada camada da rede, computando a função sigmóide (Figura 11) para todos os casos de entrada na mesma camada.

```
__global__ void forward_layer(float *d_weights, int weightOffset, int
weightsPerNeuron,
                              float *d_ins, int neuronsPrev, float *d_outs)
{
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int ixIn = blockIdx.x * neuronsPrev;
    int wid = weightOffset + (threadIdx.x * weightsPerNeuron);

    float a = d_weights[wid];

    for (int i = 1; i < weightsPerNeuron; ++i)
        a += d_weights[wid + i] * d_ins[ixIn + i-1];

    d_outs[tid] = asigmoid(a);
}
```

Figura 10: *Kernel* responsável por executar a propagação da frente por camada.

```
__device__ float asigmoid(float t)
{
    return 1.0f / (1.0f + expf(-t));
}
```

Figura 11: Função responsável por calcular o valor sigmóide.

A etapa de retropropagação inicia-se a seguir com a chamada do *kernel* responsável por calcular o valor delta para a camada de saída (Figura 12). Em seguida, uma iteração sobre as camadas anteriores é realizada para que seja calculado o valor delta para as mesmas, com a chamada de um novo *kernel*, descrito na Figura 13.

```
__global__ void deltas_output(float *outs, float *expected, float *d_deltas,
                              float *err)
{
    int tid = blockIdx.x * blockDim.x + threadIdx.x;

    err[tid] = expected[tid] - outs[tid];
    d_deltas[tid] = -err[tid] * dsigmoid(outs[tid]);
}
```

Figura 12: *Kernel* responsável por calcular o valor delta para a camada de saída.

```
__global__ void deltas_hlayer(float *outs, float *d_weights, float *d_deltas,
                              float *d_dltnext, int neuronsNext,
                              int nxtLayerWOffset, int weightsPerNeuronNxt)
{
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int oid = blockIdx.x * neuronsNext;
    int wid = nxtLayerWOffset + threadIdx.x + 1;

    d_deltas[tid] = 0.0f;
    for (int i = 0; i < neuronsNext; ++i, wid += weightsPerNeuronNxt)
        d_deltas[tid] += d_weights[wid] * d_dltnext[oid+i] * dsigmoid(outs[tid]);
}
```

Figura 13: *kernel* responsável por calcular o valor delta para a camada h.

Em uma próxima etapa, o valor da derivada do erro é calculada para os pesos em cada camada da rede. O *kernel* responsável está descrito na Figura 14. Ao fim deste passo, é realizada uma chamada ao *kernel* responsável pela atualização dos pesos (Figura 15).

```
__global__ void derivs_layer(float *d_inputs, float *d_deltas, float *d_derivs,
                             int nNeurons, int neuronsPrev,
                             int nWeights, int weightsPerNeuron, int weightOffset)
{
    int wid = blockIdx.x * nWeights + weightOffset + threadIdx.x;

    int did = blockIdx.x * nNeurons + (threadIdx.x / weightsPerNeuron);

    int iid = blockIdx.x * neuronsPrev + (threadIdx.x % weightsPerNeuron) - 1;

    float inp = (threadIdx.x % weightsPerNeuron == 0? 1.0f : d_inputs[iid]);

    d_derivs[wid] = d_deltas[did] * inp;
}
```

Figura 14: *kernel* responsável pelo cálculo do valor da derivada para cada peso por camada.

```
__global__ void update_weights_nreduc(float *d_weights, float *d_derivs, float lrate,
                                      int nCases, int nWeights)
{
    float dE = 0.0f;
    int wid = blockIdx.x * blockDim.x + threadIdx.x;

    // sum all derivs for the same weight
    for (int i = 0; i < nCases; ++i)
       dE += d_derivs[i * nWeights + wid];

    // update weight
    d_weights[wid] -= (lrate * dE);
}
```

Figura 15: *kernel* responsável por atualizar os pesos da rede neural.

Ao fim destes passos, ocorre uma nova iteração nas épocas, repetindo-se as mesmas etapas. E ao concluir as iterações sobre as épocas, o algoritmo é finalizado, e a rede neural está treinada de acordo com os dados recebidos.

O código completo da implementação proposta encontra-se no apêndice B deste trabalho.

5.2 Técnicas de Implementações Relacionadas

Dentre os trabalhos analisados durante a realização desta pesquisa, foram escolhidos dois para se realizar uma análise comparativa com a técnica de implementação proposta neste trabalho.

Os trabalhos selecionados foram o **T 07** e o **T 08**, de acordo com a tabela **Tabela 4.2**: Trabalhos selecionados. Para que se obtivesse uma análise comparativa pertinente, a escolha dos trabalhos seguiu os seguintes critérios:

1. Abordar diretamente uma técnica que implemente o treinamento da Rede Neural utilizando o algoritmo de retropropagação;
2. Utilizar o ambiente CUDA para desenvolvimento.

Para se realizar outra análise comparativa, também foi desenvolvido um algoritmo sequencial para execução na CPU. Os tópicos a seguir descrevem as técnicas de implementação propostas para os trabalhos escolhidos e para a abordagem sequencial.

5.2.1 T 07: Training and applying a feedforward multilayer neural network in GPU

O artigo apresenta a implementação de uma rede neural multicamada para execução paralela em GPU utilizando do ambiente CUDA.

A rede neural utilizada possui três camadas: a de entrada, uma camada escondida e a camada de saída. O treinamento da rede utiliza o algoritmo de retropropagação. A função de ativação utilizada é a função sigmóide.

Na fase de propagação para a frente do treinamento, são necessárias multiplicações entre matrizes e vetores. Em um primeiro passo, efetua-se o somatório entre o vetor de entrada e a matriz de pesos que ligam esta à camada escondida. Nesta etapa, a implementação CUDA utiliza do recurso matrixMul, proveniente da SDK CUDA. No segundo passo, efetua-se o cálculo da função de ativação para cada neurônio da segunda camada; os dois passos anteriores se repetem para a segunda camada e então se inicia a etapa de retropropagação em si. Primeiro calcula-se o valor do erro dos pesos para a camada de saída, utiliza-se de uma simples operação com vetores entre a diferença dos valores obtidos e os valores esperados;

durante o cálculo dos erros para segunda camada, ocorre a multiplicação entre a matriz de pesos e o vetor de erros obtido anteriormente. Em seguida, o resultado é armazenado no mesmo vetor de erros utilizado. Não foram descritas as implementações das etapas do cálculo das derivadas e da atualização dos pesos.

O treinamento ocorreu em épocas, onde treinava-se, para cada época, os casos de teste sequencialmente. Para avaliar o desempenho, variou-se a quantidade de neurônios da camada escondida entre 8, 16, 32, 64, 128, 256, 512 e 1024.

5.2.2 T 08: Parallel Training of a Back-Propagation Neural Network using CUDA

O trabalho propõe uma implementação paralela para execução em GPU utilizando CUDA e CUBLAS para o treinamento de uma rede neural com o algoritmo de retropropagação.

A estrutura da rede neural proposta possui três camadas: a de entrada, uma camada escondida, e a camada de saída. Esta última, com apenas um neurônio, e as demais possuem quantidades variadas de neurônios que são determinadas durante a execução, dependendo do caso de teste a ser utilizado.

No ambiente CUDA, em questão de memória, foram alocadas matrizes pelo CUBLAS para armazenar os valores dos pesos da camada escondida e da camada de saída. Também foi alocado um vetor para armazenar o valor das entradas. Para as matrizes e os vetores demais, a alocação foi feita pelo CUDA.

Tendo apenas uma camada escondida, pode-se determinar uma quantidade fixa de passos para o treinamento com retropropagação independentemente do caso de teste envolvido. Sendo assim, o algoritmo de treinamento possui oito passos, sendo os quatro primeiros referentes a propagação para a frente, e os quatro últimos a retropropagação:

1. Calcula-se o somatório dos pesos multiplicados pelos valores de entrada. Neste caso, o algoritmo utilizou a função cublasSgemm proveniente da biblioteca CUBLAS;
2. Aloca-se uma *thread* para calcular a função de ativação para cada neurônio da camada escondida. A função sigmóide foi utilizada como função de ativação para os neurônios da rede;
3. Novamente, a função cublasSgemm é utilizada, agora para calcular o somatório do resultado do cálculo da função de ativação para os neurônios da camada escondida, que é a saída desta camada, multiplicados pelos pesos da camada de saída;

4. Da mesma maneira que o segundo passo, aloca-se uma *thread* para efetuar o cálculo da função de ativação para o neurônio da camada de saída;
5. Inicia-se a retropropagação. Calcula-se o valor do erro para a camada de saída na CPU;
6. Aloca-se uma *thread* para calcular o valor da derivada para cada neurônio da camada escondida;
7. Os pesos da camada de saída são atualizados alocando-se uma *thread* para cada peso;
8. Os pesos da camada escondida são atualizados utilizando-se a função cublasSgemm.

O treinamento ocorre em épocas e para cada época, treina-se sequencialmente caso a caso do conjunto de dados. Para comparar o desempenho, variou-se a quantidade de neurônios da camada escondida.

5.2.3 Técnica de implementação sequencial

A implementação sequencial, desenvolvida em linguagem C, tem como objetivo promover testes e comparações com a proposta de paralelismo. Esta é uma implementação bastante conhecida na literatura.

5.2.3.1 Organização da memória

A princípio, os dados estão organizados na memória a partir de uma estrutura que armazena as características de uma camada da Rede Neural, como, por exemplo: o número de neurônios presentes na camada, ponteiros para encadear os valores dos pesos[4], as funções de ativação, as saídas de cada neurônio da camada, e mais dois ponteiros que irão representar a camada anterior e a camada posterior a mesma, sendo que, quando esta for a primeira camada, o ponteiro que indica a camada anterior terá valor nulo, e, quando esta for a última camada, terá valor nulo o ponteiro que representa a camada posterior.

[4] A estrutura adotada para alocar os valores dos pesos é baseada em um vetor que armazena todos os valores dos pesos de uma camada. Dado que, o formato padrão para este seria uma matriz, adaptou-se este para um armazenamento linear onde, para se encontrar o valor do peso w_{ji} que um neurônio n_j da camada l_c recebe do neurônio n_i da camada l_{c-1}, realiza-se no vetor de pesos $w = [w_{11}, w_{12}, \dots, w_{1i}, w_{21}, w_{22}, \dots, w_{2i}, \dots, w_{j1}, w_{j2}, \dots, w_{ji}]$ a atribuição: $w[i * (n_{neurons\_prev} + 1) + j$, onde $n_{neurons\_prev}$ corresponde ao número de neurônios da camada anterior, para a representação do valor bias, acrescenta-se a quantidade de 1 (um) a este valor, para localizar o índice correspondente.

Também possui uma estrutura que representa a Rede Neural, contendo um valor que armazena o número de camadas presentes na rede, e ponteiros que indicam quais são a camada de entrada e a camada de saída da mesma.

Uma estrutura que define o conjunto de dados que irá ser utilizado para a execução da Rede Neural, que contém atributos que armazenam o número total de casos do conjunto, o número total de entradas e de saídas dos casos e ponteiros que representam uma matriz de valores de entrada e de saída, onde as linhas indicam o caso e as colunas representam os valores de entrada ou saída.

5.2.3.2 Inicialização

Na parte de inicialização, o algoritmo oferece métodos que realizam as seguintes tarefas:

5.2.3.2.1 Criação de uma camada

Esta função recebe como entrada um ponteiro que indica qual será a camada anterior a camada a ser criada (no caso de ser a primeira camada da rede, será passado como parâmetro um valor nulo) e um valor indicando o número de neurônios que esta camada possuirá. Então, aloca-se memória em um ponteiro para definir a camada, atribui-se este ponteiro ao que indica a próxima camada da camada anterior, insere o valor do ponteiro que representa a camada anterior ao atributo que indica o mesmo no novo ponteiro criado, e ao ponteiro que indica o valor da próxima camada, atribui-se um valor nulo. Em seguida, aloca-se memória para os ponteiros que armazenam os valores de ativação e de saída de cada neurônio, baseado no valor que indica a quantidade de neurônios.

Por fim, aloca-se memória para os pesos, utilizando da informação que: o número de pesos que uma camada está associada é resultado da multiplicação entre o número de neurônios da camada atual e o número de neurônios da camada anterior, sendo que, acrescenta-se a quantidade um a este último valor para representar o peso do viés.

5.2.3.2.2 Criação de uma Rede Neural

Em um primeiro passo, aloca-se memória para referenciar a rede e se atribui a quantidade 1 (um) ao atributo que corresponde ao número de camadas. Em seguida, atribuem-se os valores que correspondem à camada de entrada da rede, inserindo-se o valor que corresponde à quantidade de neurônios da camada de entrada (que deve ser recebido como parâmetro da função), e atribuindo valores nulos aos ponteiros que correspondem aos pesos,

às funções de ativação e às camadas anterior e posterior, sendo que, ao ponteiro que indica o valor da saída do neurônio, aloca-se memória baseando-se na quantidade de neurônios contidos na camada. Na maneira em que, a princípio, a rede possui apenas uma camada, os ponteiros que referenciam a camada de entrada e de saída da mesma, apontam para a camada que foi criada no escopo desse método.

5.2.3.2.3 Adicionar uma camada na rede

Recebe-se como parâmetro de entrada uma referência para a rede na qual se deseja realizar o acréscimo e o número de neurônios que terá na camada a ser criada. Assim, utiliza-se o método, já mencionado anteriormente, responsável por criar uma camada para definir a nova camada; em seguida, atribui-se esta nova camada ao ponteiro que indica a camada de saída da rede neural, e, por fim, incrementa-se o valor que indica o número de camadas da rede.

5.2.3.2.4 Inicialização dos pesos

Os pesos da rede são inicializados com valores randômicos próximos a zero, o valor destes é pertecem à variação entre o valor máximo absoluto de 1,6, ou seja, entre -1,6 e 1,6.

5.2.3.3 Treinamento

Após a inicialização e estruturação da rede neural, faz-se necessário o treinamento da mesma. Para esta etapa, foi adotado o treinamento por lotes, onde o algoritmo itera exaustivamente sobre n épocas até que seja encontrado um momento em que se ache necessário interrompê-lo, pois o mesmo já deve ter alcançado um estado em que a atualização dos pesos não mais está sendo eficiente ou então uma quantidade de repetições que se acredite serem suficientes para o aprendizado da rede. Sendo assim, temos as seguintes funções para a execução do algoritmo:

5.2.3.3.1 Propagação para frente

Recebe-se como parâmetro para este método, a referência para a rede neural na qual o treinamento deve ocorrer, a função de ativação que deverá ser utilizada para realizar o cálculo das saídas e o ponteiro que indica os valores do conjunto de entrada. Em seguida, criam-se estruturas para representar a camada atual, recebendo como valor inicial a camada de entrada da rede, e a camada anterior, que é atribuído valor a partir do ponteiro que indica a camada anterior da camada de entrada da rede, que, neste caso, possui valor nulo.

A seguir, são atribuídos aos valores dos neurônios da camada de entrada o conteúdo do conjunto de entrada, assumindo-se que o número de neurônios na camada de entrada possui a mesma quantidade de valores de entrada que o próprio conjunto de entradas.

O treinamento efetivo irá acontecer ao iterar sequencialmente sobre as camadas presentes na rede neural. A cada iteração, sobre cada camada, a partir da camada de entrada, é calculado o valor de saída para cada neurônio, baseado no cálculo da função descrita na seção 2.1 deste presente trabalho. Em seguida, é aplicado sobre este, a função de ativação escolhida para determinar o valor final de saída do neurônio. Este também foi baseado nas equações f(a) descritas na seção referenciada anteriormente, podendo ser optada entre a função limiar e a função sigmóide.

5.2.3.3.2 Retropropagação

A matriz que aloca os referidos valores dos deltas para cada neurônio da rede segue a estrutura padrão, onde as linhas equivalem ao número de camadas da rede decrementado em 1 (um), já que a camada de entrada não possui referência para este valor, e, para as colunas, o número de neurônios em cada camada.

Na matriz que aloca os referidos valores das derivadas dos erros para cada peso da rede, as linhas correspondem ao número de camadas decrementado em 1 (um), já que a camada de entrada não possui camada anterior e, sendo assim, seus neurônios não possuem pesos. Para representação das colunas, o número de neurônios da camada atual é multiplicado pela quantidade de pesos da camada anterior incrementado em 1 (um), para corresponder ao valor de bias.

O método que realiza o cálculo dos valores delta para cada neurônio da rede tem como parâmetro de entrada as referências para a rede neural, a matriz de deltas, a saída esperada dos neurônios da camada de saída, e a função de ativação para computação dos valores. A princípio, realiza-se o cálculo do valor delta da camada de saída, e em seguida, computam-se os valores dos neurônios restantes retrocedendo na sequência das camadas.

Para o cálculo dos valores das derivadas de erro dos pesos da rede, a função recebe como parâmetro as referências para a rede neural, a matriz de deltas e a matriz de derivadas. Computa-se o valor de cada derivada seguindo o modelo citado na seção 2.3.2 deste trabalho.

5.2.3.3.3 Treinamento por lotes

Inicializam-se as matrizes que referenciam para os valores de deltas e de derivadas; para cada época, computa-se a saída para todos os neurônios da rede para todos os casos de

testes com o método de propagação para frente. Em seguida, para o passo de retropropagação, chama-se o método que calcula o valor dos deltas e o que calcula a derivada dos pesos para todos os casos de teste. Por fim, realiza-se a atualização dos pesos de todos os casos de teste baseados nos valores inseridos na matriz correspondente as derivadas de erro.

6. Experimentos e Resultados

Este capítulo expõe os dados obtidos a partir da comparação da execução das implementações apresentadas, como também uma análise comparativa de desempenho das mesmas.

Os trabalhos escolhidos para se realizar a comparação com a proposta desta pesquisa foram o T 07 e o T 08, porém a implementação destes não foi encontrada durante a pesquisa. Sendo assim, a partir das descrições encontradas nos trabalhos optou-se por desenvolvê-los, para assim realizar a análise comparativa com os mesmos. Entretanto, a descrição da implementação proposta em T 07, embora tenha uma organização que corresponda aos requisitos para se realizar a comparação, não possui detalhes suficientes para que seja implementada, o que é uma realidade incompatível para um trabalho científico, onde este deveria descrever com riqueza de detalhes a sua proposta, objetivando o bem maior da comunidade científica.

A proposta de implementação encontrada em T 08, apresenta um detalhamento mais preciso dos passos necessários para realizar a implementação. Sendo assim, desenvolveu-se esta proposta em ambiente CUDA de modo semelhante ao encontrado no trabalho. Entretanto, alguns trechos da implementação não foram descritos com detalhes suficientes pra que este fosse implementado do mesmo modo. Dessa maneira, considera-se que a implementação desenvolvida para a comparação foi baseada na implementação proposta em T 08.

As implementações foram testadas em um computador equipado com um processador Intel Core i5-750 (2.66GHz, 8Mb de memória cache), memória RAM DDR3 1333MHz de 4Gb e com a placa de vídeo GeForce GTX 550 Ti com 192 cores e com 1Gb de memória dedicada. A máquina de teste não é um computador de alto desempenho. No entanto, a comparação de desempenho entre as versões apontam para características de desempenho que são válidas para computadores de alto desempenho, e, provavelmente, favorecem ainda mais a implementação de redes neurais na GPU.

As implementações foram treinadas para realizar classificação em três conjuntos de dados com tamanhos variados, tanto em número de casos de entrada, quanto em número de

atributos por caso. O primeiro conjunto de dados, que é extremamente simples, é uma rede que executa a operação lógica do "ou exclusivo" (XOR), enquanto os outros casos foram adquiridos no UCI Machine Learning Repository[5]: o clássico conjunto de dados *íris* de R. A. Fisher e o conjunto *adult*, que fornece dados de censo. Os resultados do treinamento das redes para os três conjuntos de dados em ambas as versões de implementações podem ser visualizados na tabela a seguir:

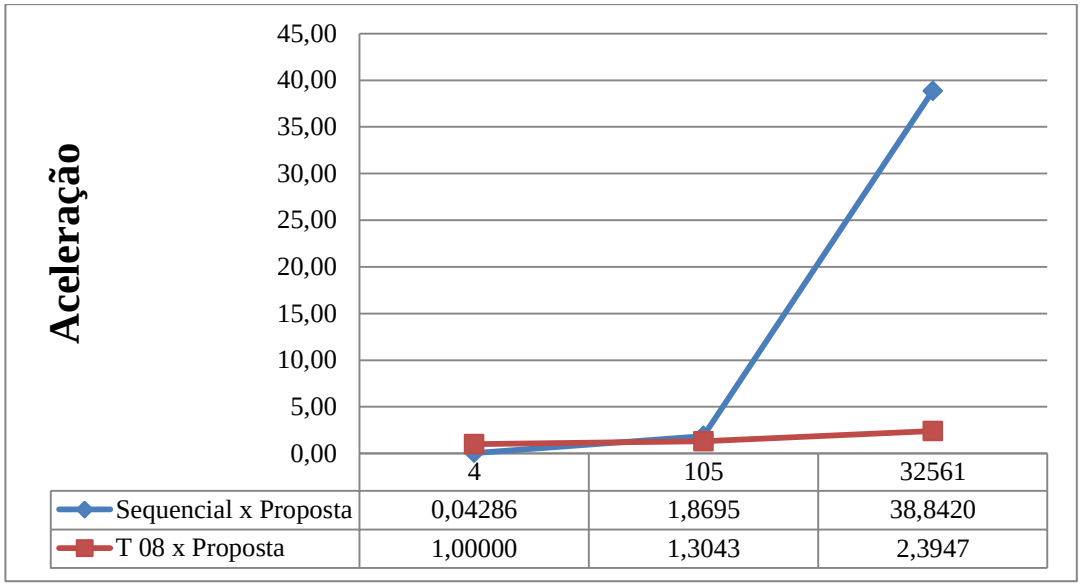
Tabela 6.1: Tempo de execução do treinamento para três conjunto de dados usando versões das implementações da técnica sequencial, da técnica proposta e da técnica encontrada em T 08.

Conjunto de dados	Quantidade de instâncias	Número de Atributos	Implementação Sequencial para CPU	Implementação T 08 para GPU	Implementação Proposta para GPU
xor	4	2	0.03s	0.7s	0.7s
iris	105	4	1.72s	1.02s	0.92s
adult	32561	14	49m12s	3m02s	1m16s

Analisando os resultados, é notável o ganho de desempenho obtido pela GPU de acordo com o aumento do tamanho do conjunto de dados. As implementações de redes neurais para GPU executam um desempenho melhor que a versão para CPU, com exceção do caso da rede XOR, com apenas quatro casos e dois atributos. Para um conjunto de dados pequeno, *íris*, as versões para GPU são cerca de duas vezes mais rápida que a versão para CPU, enquanto que, para um conjunto de dados de tamanho moderado, *adult*, as versões para GPU foram entre 16 a 38 vezes mais rápidas. Em todos os casos, a previsão de desempenho de treinamento das redes (medida por um conjunto de dados de teste com dados não relacionados com o conjunto de dados utilizados no treinamento) foi aproximadamente idêntico para as versões.

Para uma investigação mais profunda das características de desempenho entre as versões, realizou-se uma comparação de aceleração entre as versões, que pode ser visualizada na Figura 16, onde podemos observar que o desempenho da implementação proposta sobrepõe-se as demais de acordo com o aumento do número de casos presentes no conjunto de dados.

[5] http://archive.ics.uci.edu/ml/



	4	105	32561
Sequencial x Proposta	0,04286	1,8695	38,8420
T 08 x Proposta	1,00000	1,3043	2,3947

Figura 16: Comparativo entre as acelerações da implementação sequencial e da implementação T 08 sobre a implementação proposta de acordo com a quantidade de casos de entrada.

Considerando apenas as implementações para GPU, observa-se o ganho de desempenho da técnica proposta em relação à técnica encontrada em T 08 quando se comparae a execução para um conjunto de dados de mesma complexidade e variando-se apenas o número de instâncias presentes neste. Os resultados destes experimentos podem ser visualizados na Tabela 6.2. O motivo no qual ocorre esta variação no ganho de desempenho é atribuído à alternativa de paralelismo adotada pela técnica proposta, que permite treinar os casos presentes no conjunto de dados paralelamente, enquanto que na técnica em T 08, o treinamento ocorre sequencialmente para os mesmos.

Tabela 6.2: Tempo de desempenho para o treinamento da rede neural com aumento do número de casos de entrada selecionados aleatoriamente a partir do conjunto de dados *adult*. A coluna aceleração descreve quantas vezes a implementação da técnica proposta é mais rápida que a técnica apresentada em T08.

Número de instâncias	Tempo de Execução para a técnica T 08	Tempo de execução para a técnica proposta	Aceleração
10	482ms	466ms	1,0343
100	591ms	472ms	1,2521
1000	4.02s	2.67s	1,5056
10000	44.89s	24.55s	1,8285

É possível observar que a aceleração da versão sequencial sobre a implementação proposta alcança uma saturação, para esta máquina de testes, em aproximadamente 38 vezes.

A aceleração para 10 mil instâncias do conjunto de dados *adult* é cerca de 37, enquanto que para o conjunto de treinamento completo com 32 mil instâncias a aceleração foi aproximadamente 38,8. Isto indica que os *cores* de processamento da GPU estão sendo completamente utilizados, e um paralelismo mais profundo não irá melhorar significativamente o desempenho.

A linha para o tempo de execução da implementação proposta da Figura 17 mostra com clareza o lento aumento até mil instâncias, e, em seguida, um crescimento mais acentuado após o ponto de saturação. Espera-se que a mudança da GPU para uma com mais núcleos iria escalar as acelerações para conjuntos de dados ainda maiores, ou seja, a técnica de implementação proposta neste trabalho escala juntamente com o *hardware* disponível, aproveitando o poder de processamento disponibilizado pela GPU.

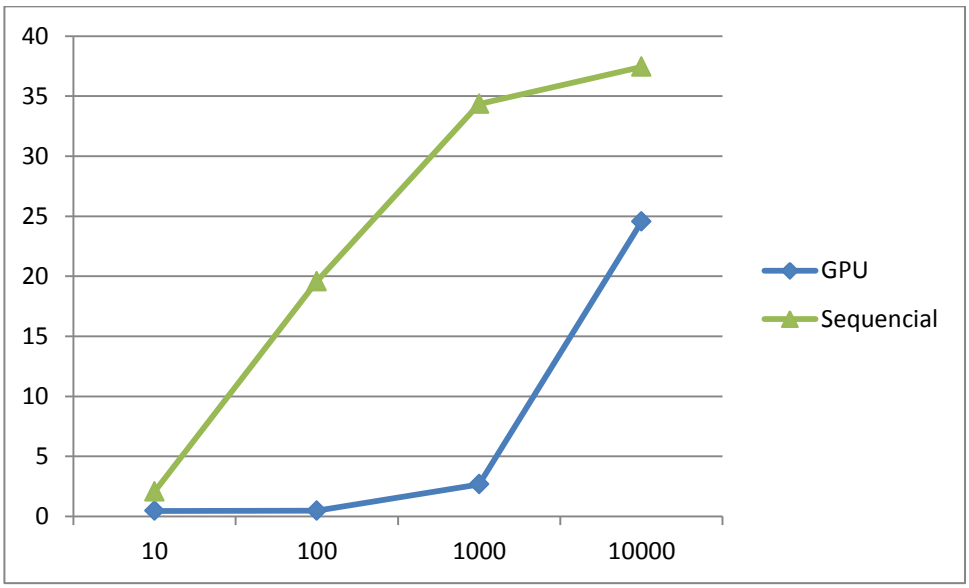


Figura 17: Tempo de execução na GPU e aceleração sobre a versão sequencial para o treinamento de redes neurais aumentando-se o número de casos de entrada selecionados aleatoriamente a partir do conjunto de dados *adult*.

7. Conclusão

O presente trabalho apresentou um conjunto de técnicas de implementação paralela para redes neurais em GPU. Tendo em vista que a arquitetura do *hardware* gráfico é muito distinta das CPUs padrões, esta exige uma estruturação diferente para implementações que almejam utilizar de seus recursos de uma maneira eficiente.

Analisando o modelo de paralelismo sugerido naturalmente pelo conceito de redes neurais, tende-se a mapear cada neurônio da rede a uma única unidade de computação. Entretanto, a dependência existente entre as camadas torna esta ideia não funcional.

O modelo de paralelismo proposto neste trabalho sugere uma paralização durante o treinamento da rede, onde se faz necessário uma repetida computação dos dados contidos nos casos de entrada. Sendo assim, a paralelização ocorreu entre as camadas da rede, onde cada camada é computada para todos os casos de entrada ao mesmo tempo, sendo as outras camadas computadas em sequência.

A técnica aqui proposta foi implementada em ambiente CUDA, e, para que esta fosse considerada relevante e eficiente, analisou-se outros modelos presentes na literatura que apresentam técnicas de implementação semelhantes a esta. Foram realizados experimentos que indicam desempenho superior da técnica proposta, tendo em vista a magnitude dos conjuntos de dados utilizados.

8. Trabalhos Futuros

O presente trabalho pode ser expandido em uma série de maneiras. Uma ideia seria utilizar outros procedimentos para otimização dos pesos em vez do treinamento com retropropagação, e ver como estes se comportam em uma implementação massivamente paralela. Temos a Aprendizagem Hebbiana e a Aprendizagem de Boltzmann como exemplos de outras técnicas de aprendizagem da rede neural (SIMON, 2001).

Outra possibilidade é a implementação de outros modelos de redes neurais. Máquinas de Boltzmann Restritas e CNN (*Convolutional Neural Networks*) têm recebido uma grande atenção de pesquisadores recentemente, devido à sua ligação com as técnicas de aprendizado profundo (KRIZHEVSKY; HINTON, 2012; SALAKHUTDINOV; HINTON, 2012; SRIVASTAVA, 2012). As suas implementações em GPUs permitiriam uma resposta mais rápida para os experimentos, e também uma avaliação de quais técnicas se encaixam melhor em *hardware* paralelo.

Outro objetivo é a implementação da técnica aqui proposta em outros ambientes de programação. Tendo em vista que foi definida uma estrutura independente da linguagem de programação utilizada, podemos reutilizá-la para outros ambientes de desenvolvimento paralelo que utilizam a GPU como, por exemplo, a arquitetura OpenCL (MUNSHI et al., 2011).

Referências Bibliográficas

AHMAD, A. et al. Multicore and GPU Parallelization of Neural Networks for Face Recognition. v. 00, 2013.

BEGOLI, E.; HOREY, J. Design Principles for Effective Knowledge Discovery from Big Data. **… on Software Architecture (ECSA), 2012 Joint …**, p. 215-218, ago. 2012.

CALDIERA, V.; ROMBACH, H. The goal question metric approach. **Encyclopedia of Software Engineering**, v. 2, p. 1-10, 1994.

CAMARGO, R. Y. DE; ROZANTE, L.; SONG, S. W. A multi-GPU algorithm for large-scale neuronal networks. n. October 2010, p. 556-572, 2011.

DOLAN, R.; DESOUZA, G. GPU-based simulation of cellular neural networks for image processing. **2009 International Joint Conference on Neural Networks**, p. 730-735, jun. 2009.

HAN, J.; KAMBER, M. **Data Mining: Concepts and Techniques**. 2nd. ed. [S.l.] Diane Cerra, 2006.

HO, T.-Y.; LAM, P.-M.; LEUNG, C.-S. Parallelization of cellular neural networks on GPU. **Pattern Recognition**, v. 41, n. 8, p. 2684-2692, ago. 2008.

KANDEL, E.; SCHWARTZ, J.; JESSELL, T. **Principles of neural science**. 4th. ed. New York: McGraw-Hill Companies, 2000.

KOCH, C. Complexity and the Nervous System. **Science**, v. 284, n. 5411, p. 96-98, 2 abr. 1999.

KRIZHEVSKY, A.; HINTON, G. E. ImageNet Classification with Deep Convolutional Neural Networks. **Neural Information Processing Systems 25**, p. 1-9, 2012.

LENT, R. et al. How many neurons do you have? Some dogmas of quantitative neuroscience under revision. **The European journal of neuroscience**, v. 35, n. 1, p. 1-9, jan. 2012.

LIU, Y. et al. Parallel implementation of neural networks training on graphic processing unit. **2012 5th International Conference on BioMedical Engineering and Informatics**, n. Bmei, p. 1571-1574, out. 2012.

MARSLAND, S. **Machine Learning: An Algoritmic Perspective**. 1st. ed. Boca Raton: Chapman & Hall/CRC, 2009.

MCCULLOCH, W. S.; PITTS, W. A logical calculus of the ideas immanent in nervous activity. **Bulletin of Mathematical Biophysics**, p. 115-133, 1943.

MUNSHI, A. et al. **OpenCL Programming Guide**. [S.l.] Pearson Education, 2011.
NVIDIA. **Compute Unified Device Architecture Programming Guide**. [S.l.] NVIDIA Corporation, 2008.

OH, K.-S.; JUNG, K. GPU implementation of neural networks. **Pattern Recognition**, v. 37, n. 6, p. 1311-1314, jun. 2004.

RAIZER, K.; IDAGAWA, H. Training and applying a feedforward multilayer neural network in GPU. **fem.unicamp.br**, p. 14, 2009.

SALAKHUTDINOV, R.; HINTON, G. A Better Way to Pretrain Deep Boltzmann Machines. **Neural Information Processing Systems 25**, n. 3, p. 1-9, 2012.

SIERRA-CANTO, X.; MADERA-RAMIREZ, F.; UC-CETINA, V. Parallel Training of a Back-Propagation Neural Network Using CUDA. **2010 Ninth International Conference on Machine Learning and Applications**, p. 307-312, dez. 2010.

SIMON, H. **Redes Neurais–Princípios e Prática**. Porto Alegre: [s.n.].
SRIVASTAVA, N. Multimodal Learning with Deep Boltzmann Machines. **Neural Information Processing Systems 25**, p. 1-9, 2012.

APÊNDICE A

Descrição do protocolo utilizado para realizar o mapeamento do estudo sistemático

Protocolo

1. Questão principal da pesquisa

A pesquisa destina-se a encontrar trabalhos que abordam técnicas de implementação de Redes Neurais para processamento paralelo em GPU. Sendo assim, questiona-se: Como implementar redes neurais em GPU?

2. Sub-questões da pesquisa

Para se obter resultados mais precisos na pesquisa, foram definidas algumas sub-questões:

Qual modelo de Rede Neural foi abordado?

Qual algoritmo foi utilizado para treinar a Rede Neural?

Qual técnica de paralelização foi proposta?

Qual ambiente de desenvolvimento foi adotado?

3. Processo de busca

A busca dos trabalhos foi realizada nas seguintes bases de dados:

- IEEE *Xplore*
- *Google Scholar*
- *Science Direct*

4. Strings de busca

Para realizar a busca formatou-se a string para que se encontrasse no título as palavras-chave: "*Neural*" ou "*Neuronal*", seguidas de "*Network*" ou "*Networks*", e também "GPU" ou "CUDA" ou "*Graphic Processing Unit*".

Entretanto, cada base de dados possui uma maneira específica de busca, logo, as strings formatadas de busca utilizadas, para cada base, foram as seguintes:

- **IEEE *Xplore*:** (("*Document Title":Neural* OR "*Document Title":Neuronal*) AND ("*Document Title":Network* OR "*Document Title":Networks*)) AND ("*Document Title"*:GPU OR "*Document Title*":CUDA OR "*Document Title":Graphic Processing Unit*)
- ***Google Scholar***: ((notítulo:*Neural* OR notítulo:*Neuronal*) AND (notítulo:*Network* OR notítulo:*Networks*)) AND (notítulo:GPU OR notítulo:CUDA OR notítulo:*Graphic Processing Unit*)
- ***Science Direct*:** ((*Title(Neural*) OR *Title(Neuronal))* AND (*Title(Network)* OR *Title(Networks*))) AND (*Title(*GPU) OR *Title*(CUDA) OR *Title(Graphic Processing Unit))*

5. Critérios de inclusão

Para que um trabalho tenha sido selecionado na pesquisa, este atendeu ao critério de inclusão definido por:

O trabalho encontrado aborda a temática: implementação de Redes Neurais paralelas?

Se SIM, inclua.

6. Critérios de exclusão

Para o critério de exclusão, foi definido:

O trabalho encontrado propõe alguma técnica de paralelização de redes neurais para GPU?

Se NÃO, exclua.

APÊNDICE B

Implementação de Redes Neurais Paralelas em CUDA.

```
/*

  mlpnnets.cu
  Implementation of feedforward MLP neural networks in CUDA.

  Andrei de A. Formiga, 2012-05-09

 */

#include <stdio.h>
#include <stdlib.h>

#include "mlpnnets.h"


// --- utility functions ---------------------------------------------------
inline float* allocateFloatsDev(int n)
{
    float *res;

    if (cudaMalloc((void**) &res, n * sizeof(float)) != cudaSuccess) {
        return NULL;
    }

    return res;
}

// --- activation functions ------------------------------------------------

// sigmoid activation function
__device__ float asigmoid(float t)
{
    return 1.0f / (1.0f + expf(-t));
}

__device__ float dsigmoid(float output)
{
    return output * (1.0f - output);
}


// --- initialization ------------------------------------------------------

// make randomly generated weights in (0.0, 1.0] be in the
// interval from -max_abs to +max_abs
__global__ void normalize_weights(float *w, float max_abs)
{
    int tid = blockIdx.x * blockDim.x + threadIdx.x;

    w[tid] = ((w[tid] - 0.5f) / 0.5f) * max_abs;
}

// random initialization for weights
// w must be an array of floats on the device
void RandomWeights(MLPNetwork *net, float max_abs, long seed)
{
    curandGenerator_t gen;
```

```
    // create and initialize generator
    curandCreateGenerator(&gen, CURAND_RNG_PSEUDO_XORWOW);
    curandSetPseudoRandomGeneratorSeed(gen, seed);
    curandSetGeneratorOrdering(gen, CURAND_ORDERING_PSEUDO_SEEDED);

    curandGenerateUniform(gen, net->d_weights, net->nWeights);
    normalize_weights<<<1, net->nWeights>>>(net->d_weights, max_abs);
    curandDestroyGenerator(gen);
}

// initialize weights randomly using the supplied generator
// w must be an array of floats on the device
void RandomWeightsGen(MLPNetwork *net, float max_abs, curandGenerator_t gen)
{
    curandGenerateUniform(gen, net->d_weights, net->nWeights);
    normalize_weights<<<1, net->nWeights>>>(net->d_weights, max_abs);
}


// --- network construction and management --------------------------------
void DestroyLayer(MLPLayer *layer)
{
    if (layer->d_outs != NULL) {
        cudaFree(layer->d_outs);
        layer->d_outs = NULL;
    }

    if (layer->d_deltas != NULL) {
        cudaFree(layer->d_deltas);
        layer->d_deltas = NULL;
    }

    free(layer);
}

// free all memory on device reserved for deltas, on all layers
void FreeDeltas(MLPNetwork *nnet)
{
    for (int i = 1; i < nnet->nLayers; ++i) {
        if (nnet->layers[i]->d_deltas != NULL) {
            cudaFree(nnet->layers[i]->d_deltas);
            nnet->layers[i]->d_deltas = NULL;
        }
    }
}

void FreeOutputs(MLPNetwork *nnet)
{
    // do not free memory for layer 0 outputs (they come from inputs)
    for (int i = 1; i < nnet->nLayers; ++i) {
        if (nnet->layers[i]->d_outs != NULL) {
            cudaFree(nnet->layers[i]->d_outs);
            nnet->layers[i]->d_outs = NULL;
        }
    }
}

// allocates memory on device for outputs on all layers
```

```
// assumes the number of cases is already net on the nnet object
bool ReallocateOutputs(MLPNetwork *nnet)
{
    // free outputs if already allocated
    FreeOutputs(nnet);

    // allocate memory for outputs (don't allocate for layer 0)
    for (int i = 1; i < nnet->nLayers; ++i) {
        nnet->layers[i]->d_outs =
            allocateFloatsDev(nnet->layers[i]->nNeurons * nnet->nCases);
        if (nnet->layers[i]->d_outs == NULL) {
            FreeOutputs(nnet);
            return false;
        }
    }

    return true;
}

// allocates memory on device for the deltas on all layers
// assumes the number of cases is already set on the nnet object
bool ReallocateDeltas(MLPNetwork *nnet)
{
    // free deltas if already allocated
    FreeDeltas(nnet);

    // allocate memory for deltas, except for input layer
    for (int i = 1; i < nnet->nLayers; ++i) {
        nnet->layers[i]->d_deltas =
            allocateFloatsDev(nnet->layers[i]->nNeurons * nnet->nCases);
        if (nnet->layers[i]->d_deltas == NULL) {
            FreeDeltas(nnet);
            return false;
        }
    }

    return true;
}

MLPLayer *CreateLayer(int nNeurons, int nNeuronsPrev, int wOffset)
{
    MLPLayer *result = (MLPLayer*) calloc(1, sizeof(MLPLayer));

    if (result == NULL)
        return NULL;

    result->nNeurons = nNeurons;

    // mark outputs and deltas as not allocated
    result->d_outs = NULL;
    result->d_deltas = NULL;

    result->weightsPerNeuron = nNeuronsPrev + 1;
    result->weightOffset = wOffset;

    return result;
}
```

```
// Create a MLP neural network for execution on the GPU.
// nLayers: number of layers
// neuronsPerLayer: array of ints (size equal to nLayers) with the
//                  number of neurons for each layer
MLPNetwork *CreateNetwork(int nLayers, int *neuronsPerLayer)
{
    MLPNetwork *result;

    result = (MLPNetwork*) calloc(1, sizeof(MLPNetwork));

    if (result == NULL)
        return NULL;

    // network is not initially prepared to store outputs, so zero input cases
    result->nCases = 0;

    result->nLayers = nLayers;
    result->layers = (MLPLayer**) calloc(nLayers, sizeof(MLPLayer*));

    if (result->layers == NULL) {
        free(result);
        return NULL;
    }

    // create input layer
    result->layers[0] = CreateLayer(neuronsPerLayer[0], 0, 0);
    if (result->layers[0] == NULL) {
        DestroyNetwork(result);
        return NULL;
    }

    // create remaining layers, and sum the number of weights
    int nwTotal = 0;
    int nwPrev = neuronsPerLayer[0];
    for (int i = 1; i < nLayers; ++i) {
        result->layers[i] = CreateLayer(neuronsPerLayer[i], nwPrev, nwTotal);
        if (result->layers[i] == NULL) {
            DestroyNetwork(result);
            return NULL;
        }

        nwTotal += neuronsPerLayer[i] * (nwPrev + 1);
        nwPrev = neuronsPerLayer[i];
    }

    result->nWeights = nwTotal;
    result->d_weights = allocateFloatsDev(result->nWeights);

    if (result->d_weights == NULL) {
        DestroyNetwork(result);
        return NULL;
    }

    return result;
}

void DestroyNetwork(MLPNetwork *net)
{
```

```
    if (net->d_weights != NULL) {
        cudaFree(net->d_weights);
        net->d_weights = NULL;
    }

    if (net->layers != NULL) {
        for (int i = 0; i < net->nLayers; ++i)
            if (net->layers[i] != NULL)
                DestroyLayer(net->layers[i]);

        free(net->layers);
        net->layers = NULL;
    }

    free(net);
}

DataSet* CreateDataSet(int nCases, int inputSize, int outputSize)
{
    DataSet *result;

    result = (DataSet*) malloc(sizeof(DataSet));

    if (result == NULL)
        return NULL;

    result->nCases = nCases;
    result->inputSize = inputSize;
    result->outputSize = outputSize;

    result->inputs = (float*) malloc(sizeof(float) * nCases * inputSize);

    if (result->inputs == NULL) {
        free(result);
        return NULL;
    }

    result->outputs = (float*) malloc(sizeof(float) * nCases * outputSize);

    if (result->outputs == NULL) {
        free(result->inputs);
        free(result);
        return NULL;
    }

    result->location = LOC_HOST;
    result->d_inputs = NULL;
    result->d_outputs = NULL;

    return result;
}

void DestroyDataSet(DataSet *dset)
{
    if (dset->inputs != NULL) {
        free(dset->inputs);
        dset->inputs = NULL;
    }
```

```
    if (dset->outputs != NULL) {
        free(dset->outputs);
        dset->outputs = NULL;
    }

    if (dset->location == LOC_HOST) {
        if (dset->d_inputs != NULL || dset->d_outputs != NULL) {
            fprintf(stderr, "Location of dataset is HOST but device ptrs are not NULL\n");
            exit(-1);
        }
    }
    else {
        if (dset->d_inputs != NULL) {
            cudaFree(dset->d_inputs);
            dset->d_inputs = NULL;
        }

        if (dset->d_outputs != NULL) {
            cudaFree(dset->d_outputs);
            dset->d_outputs = NULL;
        }
    }

    free(dset);
}

bool TransferDataSetToDevice(DataSet *data)
{
    cudaError_t e;

    if (data->location == LOC_HOST) {
        int nFloatsIn = data->nCases * data->inputSize;
        int nFloatsOut = data->nCases * data->outputSize;

        // allocate memory for dataset in device
        data->d_inputs = allocateFloatsDev(nFloatsIn);

        if (data->d_inputs == NULL)
            return false;

        data->d_outputs = allocateFloatsDev(nFloatsOut);

        if (data->d_outputs == NULL) {
            cudaFree(data->d_inputs);
            data->d_inputs = NULL;
            return false;
        }

        // copy dataset to device
        e = cudaMemcpy(data->d_inputs, data->inputs,
                       nFloatsIn * sizeof(float), cudaMemcpyHostToDevice);

        if (e != cudaSuccess) {
            fprintf(stderr, "Error copying dataset inputs from host to device: %s\n",
                    cudaGetErrorString(e));
            return false;
        }
```

```
        e = cudaMemcpy(data->d_outputs, data->outputs,
                       nFloatsOut * sizeof(float), cudaMemcpyHostToDevice);

        if (e != cudaSuccess) {
            fprintf(stderr, "Error copying dataset outputs from host to device: %s\n",
                    cudaGetErrorString(e));
            return false;
        }

        // change location specifier
        data->location = LOC_BOTH;
    }

    return true;
}


// ----------------------------------------------------------------------
// --- forward propagation ----------------------------------------------
// ----------------------------------------------------------------------

// calculate outputs of one layer, assuming the previous
// layer was already calculated; the outputs corresponding to
// all input cases are computed in parallel
//
// grid will be <<<Nc, Nn>>> for Nc input cases and Nn neurons in layer
__global__ void forward_layer(float *d_weights, int weightOffset, int weightsPerNeuron,
                              float *d_ins, int neuronsPrev, float *d_outs)
{
    // weightsPerNeuron is always = to neuronsPrev+1
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int ixIn = blockIdx.x * neuronsPrev;
    int wid = weightOffset + (threadIdx.x * weightsPerNeuron);

    // bias input
    float a = d_weights[wid];

    for (int i = 1; i < weightsPerNeuron; ++i)
        a += d_weights[wid + i] * d_ins[ixIn + i-1];

    d_outs[tid] = asigmoid(a);
}

// calculate outputs of one layer using a threshold activation,
// assuming the previous layer was already calculated; the outputs
// corresponding to all input cases are computed in parallel
//
// grid will be <<<Nc, Nn>>> for Nc input cases and Nn neurons in layer
__global__ void forward_layer_threshold(float *d_weights, int weightOffset,
                                        int weightsPerNeuron,
                                        float *d_ins, int neuronsPrev,
                                        float *d_outs)
{
    // weightsPerNeuron is always = to neuronsPrev+1
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int ixIn = blockIdx.x * neuronsPrev;
    int wid = weightOffset + (threadIdx.x * weightsPerNeuron);
```

```
    // bias input
    float a = d_weights[wid];

    for (int i = 1; i < weightsPerNeuron; ++i)
        a += d_weights[wid + i] * d_ins[ixIn + i-1];

    d_outs[tid] = (a > 0.0f? 1.0f : 0.0f);
}

// present a vector of input cases to the network nnet and do forward propagation.
// the dataset is assumed to contain a number of cases equal to
// the nCases in the network
void PresentInputsFromDataSet(MLPNetwork *nnet, DataSet *dset, int actf)
{
    // transfer data to device (if it's not already there)
    if (!TransferDataSetToDevice(dset)) {
        fprintf(stderr, "Could not transfer data set to device\n");
        return;
    }

    // do forward propagation
    PresentInputs(nnet, dset->d_inputs, actf);
}

// present a vector of input cases to the network nnet and do forward propagation.
// d_inputs is assumed to be in device memory, and of size equal to N * nnet->nCases,
// where N is the number of inputs to the network
void PresentInputs(MLPNetwork *nnet, float *d_inputs, int actf)
{
    nnet->layers[0]->d_outs = d_inputs;

    int nn;
    for (int l = 1; l < nnet->nLayers; ++l) {
        nn = nnet->layers[l]->nNeurons;
        if (actf == ACTF_THRESHOLD)
            forward_layer_threshold<<<nnet->nCases, nn>>>(nnet->d_weights,
                                                nnet->layers[l]->weightOffset,
                                                nnet->layers[l]->weightsPerNeuron,
                                                nnet->layers[l-1]->d_outs,
                                                nnet->layers[l-1]->nNeurons,
                                                nnet->layers[l]->d_outs);
        else
            forward_layer<<<nnet->nCases, nn>>>(nnet->d_weights,
                                                nnet->layers[l]->weightOffset,
                                                nnet->layers[l]->weightsPerNeuron,
                                                nnet->layers[l-1]->d_outs,
                                                nnet->layers[l-1]->nNeurons,
                                                nnet->layers[l]->d_outs);
    }

}

bool PrepareForTesting(MLPNetwork *nnet, int nCases)
{
    if (nnet->nCases != nCases) {
        nnet->nCases = nCases;
        return ReallocateOutputs(nnet);
```

```
    }

    // no need to reallocate outputs
    return true;
}

// ---------------------------------------------------------------------
// --- backpropagation -------------------------------------------------
// ---------------------------------------------------------------------

// Calculate the deltas for each neuron in the output layer, and the
// error between the actual and expected outputs
//
// grid should be <<<Nc, Nn>>> for Nc cases and Nn neurons in layer
__global__ void deltas_output(float *outs, float *expected, float *d_deltas,
                              float *err)
{
    int tid = blockIdx.x * blockDim.x + threadIdx.x;

    err[tid] = expected[tid] - outs[tid];
    d_deltas[tid] = -err[tid] * dsigmoid(outs[tid]);
}

// Calculate the deltas for each neuron in a hidden layer
//
// grid should be <<<Nc, Nn>>> for Nc cases and Nn neurons in layer
__global__ void deltas_hlayer(float *outs, float *d_weights, float *d_deltas,
                              float *d_dltnext, int neuronsNext,
                              int nxtLayerWOffset, int weightsPerNeuronNxt)
{
    // index for delta being calculated on hidden layer
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    // index for first delta on next layer
    int oid = blockIdx.x * neuronsNext;
    // index for relevant weights (neurons in next layer)
    int wid = nxtLayerWOffset + threadIdx.x + 1;  // +1 to account for bias weight

    d_deltas[tid] = 0.0f;
    for (int i = 0; i < neuronsNext; ++i, wid += weightsPerNeuronNxt)
        d_deltas[tid] += d_weights[wid] * d_dltnext[oid+i] * dsigmoid(outs[tid]);
}


// Calculate the derivatives of the error relative to each weight
//
// grid should be <<<Nc, Nw>>> for Nc cases and Nw total weights for layer
__global__ void derivs_layer(float *d_inputs, float *d_deltas, float *d_derivs,
                 int nNeurons, int neuronsPrev,
                 int nWeights, int weightsPerNeuron, int weightOffset)
{
    // weight index
    int wid = blockIdx.x * nWeights + weightOffset + threadIdx.x;
    // delta index
    int did = blockIdx.x * nNeurons + (threadIdx.x / weightsPerNeuron);
    // input index
    int iid = blockIdx.x * neuronsPrev + (threadIdx.x % weightsPerNeuron) - 1;

    float inp = (threadIdx.x % weightsPerNeuron == 0? 1.0f : d_inputs[iid]);
```

```
    d_derivs[wid] = d_deltas[did] * inp;
}

//__global__ sum_derivs(float *d_derivs)
//{
//}

// launch grid: <<<1, NWEIGHTS>>> for number of weights
// TODO: do a proper reduction instead of a set number of sums
__global__ void update_weights_nreduc(float *d_weights, float *d_derivs, float lrate,
                                      int nCases, int nWeights)
{
    float dE = 0.0f;
    int wid = blockIdx.x * blockDim.x + threadIdx.x;

    // sum all derivs for the same weight
    for (int i = 0; i < nCases; ++i)
    dE += d_derivs[i * nWeights + wid];

    // update weight
    d_weights[wid] -= (lrate * dE);
}

bool PrepareForTraining(MLPNetwork *nnet, DataSet *trainData)
{
    if (trainData->nCases != nnet->nCases) {
        nnet->nCases = trainData->nCases;
        if (!ReallocateOutputs(nnet))
            return false;

        if (!ReallocateDeltas(nnet)) {
            FreeOutputs(nnet);
            return false;
        }
    }

    return true;
}

float BatchTrainBackprop(MLPNetwork *nnet, DataSet *data, int epochs,
                         float lrate, int calcSSE, int printSSE)
{
    float *err = NULL, *d_err = NULL;
    float *d_derivs;
    float sse = 0.0f;
    MLPLayer *outLayer = nnet->layers[nnet->nLayers - 1];
    int nOutputs = outLayer->nNeurons;

    if (!PrepareForTraining(nnet, data))
        return -1.0f;

    if (!TransferDataSetToDevice(data))
        return -1.0f;

    // allocate space for errors
    d_err = allocateFloatsDev(nOutputs * data->nCases);
```

```
if (d_err == NULL) {
    fprintf(stderr, "Error allocating memory on device to store errors: %s\n",
            cudaGetErrorString(cudaGetLastError()));
    exit(-1);
}

if (calcSSE) {
    err = (float*) malloc(nOutputs * data->nCases * sizeof(float));
    if (err == NULL) {
        fprintf(stderr, "Couldn't allocate memory to store errors\n.");
        exit(-1);
    }
}

// allocate memory for derivatives
d_derivs = allocateFloatsDev(data->nCases * nnet->nWeights);

for (int e = 0; e < epochs; ++e) {
//printf("- Epoch %d\n", e);

    // forward propagation of all the cases
    PresentInputs(nnet, data->d_inputs, ACTF_SIGMOID);
    //cudaThreadSynchronize();

    // // print outputs (debug)
    // float *outs = (float*) malloc(data->nCases * nOutputs * sizeof(float));
    // CopyNetworkOutputs(nnet, outs);
    // for (int i = 0; i < data->nCases * nOutputs; ++i)
    //     printf("%5.3f ", outs[i]);
    // printf("|| "); printf("\n");
    // free(outs);
    // // print outputs (debug end)

    // backpropagation: calculation of deltas
    deltas_output<<<data->nCases, nOutputs>>>(outLayer->d_outs,
                                              data->d_outputs,
                                              outLayer->d_deltas,
                                              d_err);

    // // print deltas for output layer (debug)
    // float *deltas = (float*) malloc(data->nCases * nOutputs * sizeof(float));
    // cudaMemcpy(deltas, outLayer->d_deltas, data->nCases * nOutputs * sizeof(float),
    cudaMemcpyDeviceToHost);
    // for (int i = 0; i < data->nCases * nOutputs; ++i)
    //     printf("%5.3f ", deltas[i]);
    // printf(" -- ");
    // free(deltas);
    // // (debug end)


    MLPLayer *layer;
    MLPLayer *nextLayer = outLayer;
    for (int l = nnet->nLayers-2; l > 0; --l) {
        layer = nnet->layers[l];
        deltas_hlayer<<<data->nCases, layer->nNeurons>>>(layer->d_outs,
                                                         nnet->d_weights,
                                                         layer->d_deltas,
                                                         nextLayer->d_deltas,
```

```
                                                    nextLayer->nNeurons,
                                                    nextLayer->weightOffset,
                                                    nextLayer->weightsPerNeuron);
        nextLayer = layer;

        // // print deltas for layer (debug)
        // deltas = (float*) malloc(data->nCases * layer->nNeurons * sizeof(float));
        // cudaMemcpy(deltas, layer->d_deltas, data->nCases * layer->nNeurons *
        sizeof(float), cudaMemcpyDeviceToHost);
        // for (int i = 0; i < data->nCases * layer->nNeurons; ++i)
        //     printf("%5.3f ", deltas[i]);
        // printf(" -- ");
        // free(deltas);
        // // (debug end)
    }

    // calculate SSE for this epoch
if (calcSSE) {
    sse = 0.0f;
    cudaMemcpy(err, d_err, data->nCases * nOutputs * sizeof(float),
    cudaMemcpyDeviceToHost);
    for (int i = 0; i < data->nCases * nOutputs; ++i) {
    //printf("%6.3f ", err[i]);
    sse += (err[i] * err[i]);
    }

    if (printSSE)
    printf("- SSE = %5.3f\n", sse);
}

    // calculate derivatives of the error
    MLPLayer *prevLayer = nnet->layers[0];
    int nw;
    for (int l = 1; l < nnet->nLayers; ++l) {
        layer = nnet->layers[l];
        nw = layer->nNeurons * layer->weightsPerNeuron;
        derivs_layer<<<data->nCases, nw>>>(prevLayer->d_outs,
                                           layer->d_deltas,
                                           d_derivs,
                                           layer->nNeurons,
                                           prevLayer->nNeurons,
                                           nnet->nWeights,
                                           layer->weightsPerNeuron,
                                           layer->weightOffset);
        prevLayer = layer;
    }

    // update weights based on derivatives
    update_weights_nreduc<<<1, nnet->nWeights>>>(nnet->d_weights, d_derivs,
                                                 lrate, data->nCases,
                                                 nnet->nWeights);
}

if (err != NULL)
    free(err);

// cleanup
cudaFree(d_err);
```

```
    cudaFree(d_derivs);
    FreeDeltas(nnet);

    cudaThreadSynchronize();
    return sse;
}

// ------------------------------------------------------------------------
// --- utility functions ------------------------------------------------
// ------------------------------------------------------------------------

// Copy the outputs for network nnet, stored in device memory, to
// host memory pointed to by outs. outs must have size equal to N * nnet->nCases,
// where N is the number of output neurons in the network
bool CopyNetworkOutputs(MLPNetwork *nnet, float *outs)
{
    cudaError_t e;

    MLPLayer *last = nnet->layers[nnet->nLayers-1];

    e = cudaMemcpy(outs, last->d_outs,
                   last->nNeurons * nnet->nCases * sizeof(float),
                   cudaMemcpyDeviceToHost);

    if (e != cudaSuccess) {
        fprintf(stderr, "Error copying outputs from device to host: %s\n",
                cudaGetErrorString(e));
        return false;
    }

    return true;
}

void PrintWeights(MLPNetwork *nnet)
{
    float *h_weights;
    cudaError_t e;

    h_weights = (float*) malloc(nnet->nWeights * sizeof(float));

    if (h_weights == NULL) {
        printf("Error allocating host memory to copy weights.\n");
    }
    else {
        e = cudaMemcpy(h_weights, nnet->d_weights, nnet->nWeights * sizeof(float),
                       cudaMemcpyDeviceToHost);

        if (e != cudaSuccess) {
            fprintf(stderr, "Error copying weights from device to host: %s\n",
                    cudaGetErrorString(e));
        }

        for (int i = 0; i < nnet->nWeights; ++i) {
            printf("%4.5f ", h_weights[i]);
        }
        printf("\n");
    }
```

```
    free(h_weights);
}

// return an array of floats with the outputs for layer with index ixLayer
float *GetLayerOutputs(MLPNetwork *nnet, int ixLayer)
{
    int   length = nnet->layers[ixLayer]->nNeurons * nnet->nCases;
    float *result = (float*) malloc(length * sizeof(float));

    if (result == NULL)
        return NULL;

    // TODO: check cudaMemcpy for errors
    cudaMemcpy(result, nnet->layers[ixLayer]->d_outs,
               length * sizeof(float), cudaMemcpyDeviceToHost);

    return result;
}
```