UNIVERSIDADE ESTADUAL PAULISTA

Júlio de Mesquita Filho

Pós-Graduação em Ciência da Computação

Adriana Sayuri Iwashita

Implementação do Algoritmo de Treinamento do Classificador Floresta de Caminhos Ótimos em GPU

UNESP

2013

Adriana Sayuri Iwashita

Implementação do Algoritmo de Treinamento do Classificador Floresta de Caminhos Ótimos em GPU

Prof. Dr. João Paulo Papa (Orientador)

Prof. Dr. Alexandro José Baldassin (Coorientador)

Dissertação de Mestrado elaborada junto ao Programa de Pós-Graduação em Ciência da Computação – Área de Concentração em Sistemas de Computação, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação

UNESP

2013

Iwashita, Adriana Sayuri
Implementação do algoritmo de treinamento do classificador floresta de caminhos ótimos em GPU / Adriana Sayuri Iwashita, 2013
42 f. : il.

Orientador: João Paulo Papa
Coorientador: Alexandro José Baldassin

Dissertação (mestrado) - Universidade Estadual Paulista "Júlio de Mesquita Filho". Faculdade de Ciências, Bauru, 2013

1. Reconhecimento de padrões. 2. Floresta de caminhos ótimos. 3. GPU. I. Universidade Estadual Paulista "Júlio de Mesquita Filho". Faculdade de Ciências. II. Título.

CDU – 518.72

Adriana Sayuri Iwashita

Implementação do Algoritmo de Treinamento do Classificador Floresta de Caminhos Ótimos em GPU

Dissertação apresentada como parte dos requisitos para obtenção do título de Mestre em Ciência da Computação, junto ao Programa de Pós-Graduação em Ciência da Computação, Área de Concentração - Sistemas de Computação, do Instituto de Biociências, Letras e Ciências Exatas da Universidade Estadual Paulista "Júlio de Mesquita Filho", Campus de São José do Rio Preto.

Comissão Examinadora

Prof. Dr. João Paulo Papa
UNESP – Bauru
Orientador

Prof. Dr. Antonio Carlos Sementille
UNESP – Bauru

Prof. Dr. Alexandre Luis Magalhães Levada
UFSCar – São Carlos

São José do Rio Preto
15 de maio de 2013

Dedico este trabalho

Aos meus pais, por todo o apoio e educação que me proporcionaram e que me permitiu chegar até aqui.

Agradecimentos

Agradeço primeiramente à Deus, por ser o apoio que tanto busquei em todos os momentos da vida, me dando força e coragem para enfrentar as dificuldades.

Agradeço ao meu pai *Kussuo Iwashita*, que me ensinou que sempre há um lado bom nas coisas, e que mesmo nos momentos difíceis não devo me deixar abater. À minha mãe *Regina Remico Suzuki Iwashita*, que pelos seus atos me ensina como ter um bom caráter. MUITO OBRIGADA! Amo vocês.

Agradeço ao meu orientador Prof. *João Paulo Papa*, por todo o apoio e incentivo que me proporcionou durante esses anos, por ter aceito ser meu orientador tanto na graduação como na pós-graduação, e por ser o professor amigo que é. Agradeço também ao coorientador Prof. *Alexandro José Baldassim*.

Agradeço a UNESP e servidores por possibilitar a realização do mestrado e a FAPESP por conceder a bolsa durante dois anos de projeto. Agradeço também aos demais professores, por contribuir com o meu ensino; em especial ao prof. Antônio Carlos Sementille e ao prof. Alexandre Levada pelas críticas e sugestões para o trabalho.

Agradeço ao meu namorado *Bruno Keith Kochimizu*, que esteve sempre ao meu lado me apoiando em tudo, tanto nas áreas técnicas quanto emocionalmente. Obrigada pela paciência, te amo.

Agradeço ao grupo do LCAD/RECOGNA: *Mizobe*, *Luis Cláudio*, *Rafael*, *Jéssica*, *Alan*, *Clayton*, *Luis Augusto*, ... , e em especial ao *Marcos* que me ajudou muito na última fase do projeto.

Agradeço também aos meus amigos e familiares que sempre torceram pelo meu sucesso.

Enfim, agradeço a todos que me ajudaram direta ou indiretamente e pelos que torceram por mim!

Muito obrigada!

Se cheguei até aqui foi porque me apoiei no ombro dos gigantes.

Isaac Newton

Resumo

Técnicas de reconhecimento de padrões têm como principal objetivo classificar um conjunto de amostras baseadas em um conhecimento *a priori* ou em alguma informação estatística obtida dessas amostras. Tal processo de aprendizado é a fase de maior consumo de tempo na grande maioria das técnicas de reconhecimento de padrões. O problema ainda pode piorar em ferramentas de classificação interativas, nas quais o usuário é solicitado a rotular amostras que serão utilizadas para o treinamento, e após a classificação, os resultados podem ser refinados através de mais amostras rotuladas manualmente. Esta situação pode ser inaceitável para grandes bases de dados. Dado que muitos trabalhos tem sido orientados à implementação de vários algoritmos de reconhecimento de padrões em ambiente *General Purpose Graphics Processing Unit* - GPGPU, o presente estudo objetivou a implementação da etapa de treinamento do classificador Floresta de Caminhos Ótimos em *Compute Unified Device Architecture* - CUDA visando aumentar a sua eficiência. Foi implementada uma otimização do referido classificador utilizando os métodos tradicionais, ou seja, na *Central Processing Unit* - CPU, e demonstrou uma fase de treinamento cerca de duas vezes mais rápida que a versão original. A otimização do classificador em CUDA também demonstrou uma fase de treinamento mais rápida que a versão original.

Palavras-chave: *Reconhecimento de padrões*. *Floresta de Caminhos Ótimos*. *GPU*.

Abstract

Pattern recognition techniques have as main objective to classify a set of samples based on a priori knowledge or statistical information obtained by these samples. This learning process is the most time-consuming phase in most pattern recognition techniques. The problem may become worse in interactive classification tools, in which the user is asked to label the samples that will be used for training, and after the classification, the results can be refined through more samples manually labeled. However, this may be unacceptable for large databases. Since many studies have been oriented to the implementation of various pattern recognition algorithms on General Purpose Graphics Processing Unit - GPGPU environment, this study aimed the implementation of the training stage of the Optimum-Path Forest classifier in Compute Unified Device Architecture - CUDA in order to increase its efficiency. We have implemented an optimization of that classifier using the traditional methods, i.e., on the Central Processing Unit - CPU, and it has demonstrated a training phase about two times faster than the original version. The classifier optimization in CUDA has also shown a training phase faster than the original version.

Keywords: Pattern Recognition. Optimum-Path Forest. GPU.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

ANN	*Artificial Neural Networks*
CPU	*Central Processing Unit*
CTM	*Close To Metal*
CUDA	*Compute Unified Device Architecture*
GPGPU	*General Purpose Graphics Processing Unit*
GPU	*Graphics Processing Unit*
IFT	*Image Forest Transform*
MIMD	*Multiple Instruction, Multiple Data*
MISD	*Multiple Instruction, Single Data*
MSF	*Minimum Spanning Forest*
MST	*Minimum Spanning Tree*
OPF	*Optimum-Path Forest*
OPT	*Optimum-Path Tree*
SIMD	*Single Instruction, Multiple Data*
SISD	*Single Instruction, Single Data*
STL	*Standard Template Library*
SVM	*Suport Vector Machines*

1 Introdução

Técnicas de reconhecimento de padrões têm como principal objetivo classificar um conjunto de amostras (padrões) baseadas em um conhecimento *a priori* ou em alguma informação estatística obtida dessas amostras [4]. Os padrões a serem classificados são, geralmente, grupos de características extraídas de um conjunto de observações, definindo pontos em um espaço multidimensional apropriado. A tarefa de classificação das amostras é geralmente baseada em um aprendizado anterior, onde tenta-se identificar o comportamento estatístico dos dados em um conjunto de treinamento, sendo que o posterior reconhecimento deles é avaliado em um conjunto de teste. Os métodos de classificação podem ser ainda divididos em supervisionados, não-supervisionados e semi-supervisionados, de acordo com seus algoritmos de aprendizado. Métodos não-supervisionados não possuem ou não exploram conhecimento algum sobre as classes das amostras de treinamento, enquanto esta informação é conhecida e explorada em métodos supervisionados. Métodos semi-supervisionados conhecem apenas parte ou fazem uso parcial dos rótulos das amostras de treinamento.

O aprendizado de um classificador é o que mais consome esforço computacional. Técnicas tradicionais e amplamente utilizadas como, por exemplo, Redes Neurais Artificiais (*Artificial Neural Networks* - ANNs) [5], Máquinas de Vetores de Suporte (*Support Vector Machines* - SVMs) [6] e classificadores Bayesianos [4] tentam ajustar seus parâmetros ou fazem suposições a respeito da forma e/ou distribuição das amostras de treinamento no espaço de atributos com o intuito de minimizar o erro de classificação. A disposição das amostras influencia diretamente o desempenho de um classificador, o qual pode ser visto, basicamente, como uma função que tenta encontrar hiperplanos que melhor dicotomizam o espaço de atributos. Situações mais simples, também chamadas de problemas separáveis (classes com rótulos distintos podem ser separadas através de hiperplanos), podem ser facilmente resolvidas por classificadores baseados em ANNs, tais como ANNs com perceptron multicamadas (*ANN with Multilayer Perceptron* - ANN-MLP), ANNs com funções de ativação de base radial (*ANN with Radial Basis Function* - ANN-RBF) e Mapas Auto-Organizáveis de Kohonen (*Self Organizing Maps* - SOM), por exemplo. Classificadores bayesianos também conseguem endereçar tais problemas [5].

Recentemente, uma nova metodologia para o desenvolvimento de classificadores baseados em floresta de caminhos ótimos (*Optimum-Path Forest* - OPF) foi apresentada [7]. Os classificadores baseados em OPF reduzem o problema de reconhecimento de padrões ao particionamento de um grafo em árvores de caminhos ótimos, as quais são enraizadas por amostras protótipos e qualquer elemento pertencente a uma dada árvore é mais fortemente conexo à sua raiz do que qualquer outra. Tal força de conexidade é estabelecida por uma função de custo de caminho, previamente escolhida. Atualmente, duas variantes foram propostas: (i) a versão não supervisionada [8] e a (ii) versão supervisionada, a qual é subdividida em OPF com grafo

completo [7] e OPF com grafo k-vizinhos mais próximos (k-*Nearest Neighbors* – k-NN) [9], sendo a que faz uso do grafo completo a mais utilizada e difundida.

Ultimamente, empresas produtoras de placas de vídeo, tais como NVIDIA e ATI, empenharam-se em produzir placas gráficas com GPUs (*Graphic Processor Unit*) cada vez mais poderosas, ocasionando em um crescente aumento na programação orientada aos processadores gráficos, os quais são projetados especialmente para executar operações em paralelo, possibilitando, assim, um notável aumento no desempenho quando comparados às CPUs. Operações habituais em um microprocessador comum, tais como operações com pontos flutuantes, por exemplo, podem ser executadas e, principalmente, programadas agora em GPUs com mais facilidade do que antigamente. Isso deve-se ao fato de as empresas produtoras de placas de vídeo terem atentado ao fato que, após tanta pesquisa no desenvolvimento de placas gráficas, as mesmas estão tão (ou mais) poderosas quanto os tradicionais microprocessadores. Desta forma, isso motivou o fato de tais empresas encabeçarem projetos para o desenvolvimento ágil de programas para serem executados no processador da própria placa gráfica, ou seja, na GPU. Existem, basicamente, duas principais arquiteturas para desenvolvimento de aplicações em processadores gráficos: CUDA (*Compute Unified Device Architecture*) e CTM (*Close To Metal*). Enquanto CUDA é orientada para placas gráficas da NVIDIA, a segunda segue os padrões da empresa ATI, sendo a primeira delas a mais difundida.

Recentemente, diversos trabalhos orientados à implementação de técnicas de reconhecimento de padrões em GPU foram desenvolvidos, tais como implementações de redes neurais e SVMs orientadas aos processadores gráficos. Catanzaro et al. [10], por exemplo, propuseram a implementação das SVMs em GPU, a qual obteve um ganho de 9–35$\times$ de velocidade quando comparada à execução das SVMs em uma CPU tradicional. Do et al. [11] também propuseram a utilização das SVMs em GPU, porém empregaram uma abordagem paralela para implementação do classificador SVMs. Oh e Jung [12] introduziram as redes neurais no contexto de programação em GPUs. Posteriormente, Jang et al. [13] propuseram a implementação de redes neurais combinando CUDA e OpenMP (*Open Multi-Processing*), uma biblioteca para programação concorrente. Neste trabalho, os autores enfatizam a problemática da programação massiva em GPUs, ou seja, alguns cuidados devem ser tomados antes de portar técnicas para a programação orientada a placas gráficas. Uma das regras mais importantes reside no fato de que os algoritmos a serem portados para ambientes de programação concorrentes necessitam, primordialmente, possuir uma quantidade considerável de módulos que podem ser executados em paralelo. Caso contrário, o esforço computacional exigido para a troca de dados entre os processadores (*throughput*) acaba tornando tais implementações não muito vantajosas. A motivação para a implementação dessas abordagens em GPU reside no fato de elas possuírem uma fase de treinamento muito custosa. Assim, essas tradicionais abordagens tornam-se inviáveis para aplicações com grandes bases de dados, ou mesmo naquelas onde o conjunto de dados não é tão grande, mas o treinamento dos mesmos precisa ser realizado constantemente.

Nesse contexto, o presente trabalho objetiva o estudo do algoritmo de treinamento supervisionado do classificador OPF e do uso da GPU para programação de propósito geral, visando a implementação do algoritmo de treinamento do OPF no ambiente CUDA para execução em GPUs, para que no futuro seja contruída a cudaOPF, uma biblioteca de desenvolvimento de classificadores de padrões baseados em floresta de caminhos ótimos em ambiente CUDA.

O presente trabalho está dividido da seguinte maneira: o Capítulo 2 trata de alguns conceitos sobre arquitetura paralela. O Capítulo 3 trata do referencial teórico e conceitos básicos sobre o classificador OPF. O Capítulo 4 trata dos algoritmos propostos, que são descritos nas seções 4.1 e 4.2, sendo as propostas de otimização para a fase de treinamento do algoritmo OPF em CPU e em GPU respectivamente. Finalmente, as conclusões são descritas no Capítulo 5.

2 Arquiteturas Paralelas

Uma arquitetura paralela fornece uma estrutura explícita e de alto nível para o desenvolvimento de soluções utilizando o processamento paralelo, através da existência de processadores que cooperam para resolver problemas através de execução concorrente [14]. A sua ideia consiste em dividir uma aplicação de modo que possa ser executada por vários elementos de processamento que cooperam entre si, geralmente para buscar um maior desempenho e aumentar a eficiência na execução das instruções.

Outros fatores que levaram ao desenvolvimento da computação paralela foram que, devido a característica de configuração modular, determinadas aplicações podem ter vantagens nesse tipo de arquitetura e ser implementadas mais facilmente. Além disso, a evolução na redução do tamanho dos componentes é uma propriedade que possui um limite físico, sendo necessária uma alternativa para o aumento de velocidade de processamento. Também podem apresentar uma melhor relação custo/benefício quando comparadas à supercomputadores e serem tolerantes à falhas.

A cooperação dos elementos de processamento de uma arquitetura paralela é realizada através de comunicação e sincronismo, o que gera certa complexidade, e estes devem ser implementados de modo a não atrapalhar o desempenho da execução paralela.

O paralelismo pode ser realizado de duas formas: o paralelismo lógico (ou pseudoparalelismo), em que um único processador executa em cada intervalo de tempo um trecho de um dos diferentes processos que estão aguardando processamento; e o paralelismo físico, quando há mais de um processo sendo executado no mesmo intervalo de tempo em diferentes processadores.

O paralelismo físico temporal (*pipeline*) consiste na execução de subtarefas em estágios diferentes em um mesmo intervalo de tempo: uma tarefa é subdividida em uma sequência de subtarefas, e cada uma delas é executada por um estágio especializado do hardware e software, que opera concorrentemente com os outros estágios do *pipeline*. Já no *pipeline* chamado superescalar, as instruções podem executar no mesmo estágio do *pipeline* ao mesmo tempo. A quantidade de instruções que podem ser executadas em paralelo depende da quantidade e do tipo das unidades paralelas disponíveis.

O nível de paralelismo, chamado de granulação, relaciona o tamanho do que vai ser processado submetidos aos processadores. Pode ser dividida em três níveis: grossa, média e fina. A granulação grossa relaciona o paralelismo em nível de programas e processos executados ou sincronizados com maior frequência, geralmente em plataformas com poucos processadores. A granulação fina relaciona paralelismo em nível de instruções ou operações executados ou sincronizados com menor frequência, e implica em um grande número de processadores pequenos e simples [15]. A granulação média situa-se entre a granulação grossa e a fina.

Para verificar a qualidade de algoritmos paralelos, foram criados duas medidas: *speedup* e eficiência. O *speedup* verifica o aumento de velocidade quando se executa um determinado processo em p processadores em relação à execução deste processo em um processador. A eficiência trata da relação entre o *speedup* e o número de processadores. Alguns fatores que influenciam negativamente o *speedup* podem ser: sobrecarga de comunicação entre processadores, partes do código que não podem ser paralelizados (estritamente sequenciais) e o nível de paralelismo utilizado.

2.1 Taxonomia Flynn

A taxonomia introduzida por Flynn em 1972 é uma forma de classificar sistemas de processamento paralelo, baseada no fluxo de instruções e no fluxo de dados. Considera então dois fatores: o tipo de arquitetura paralela e o tipo de comunicação dos processadores. Portanto, é uma taxonomia baseada no fluxo único ou múltiplo de instruções e de dados [14]:

- *SISD* (*single instruction, single data*) - define o computador serial, baseado no modelo de von Neumann.
- *MISD* (*multiple instruction, single data*) - pode envolver múltiplos processadores aplicando diferentes instruções em um mesmo dado, não é muito utilizado na prática.
- *MIMD* (*multiple instruction, multiple data*) - envolve múltiplos processadores executando de forma autônoma diversas instruções em diferentes conjunto de dados.
- *SIMD* (*single instruction, multiple data*) - envolve múltiplos processadores executando simultaneamente a mesma instrução em diferentes conjunto de dados.

As GPUs tem uma arquitetura similar ao SIMD, em que uma única instrução pode controlar múltiplos elementos de processamento.

2.2 Unidade de Processamento Gráfico de Propósito Geral

Acompanhando a Lei de Moore, a indústria aumenta a velocidade das CPUs inserindo um número maior de transistores nos processadores. Porém, devido aos limites físicos, o aumento da velocidade aumenta o consumo de energia e a dissipação de calor. Nesse contexto, um novo paradigma foi se formando: a computação evoluindo de "processamento central" na CPU para "co-processamento" na CPU e GPU. A performance de GPUs cresce mais rápido em relação às CPUs devido, principalmente, às características de sua arquitetura paralelizada. As GPUs, inicialmente desenvolvidas com o propósito de renderização gráfica, atualmente tem um potencial enorme na área de alta performance.

As arquiteturas multinúcleos proporcionam alto poder computacional a custos baixos se comparadas a sistemas especialistas dedicados a problemas de alto desempenho. Enquanto a arquitetura das CPUs têm dois ou quatro núcleos, a arquitetura das GPUs apresentam centenas de núcleos que executam milhares de *threads*.

O uso das GPUs para programação de propósito geral (GPGPU - *General-Purpose on Graphic Processor Unit*), é uma técnica que utiliza a GPU para realizar cálculos em aplicações tradicionalmente tratadas pela CPU. A computação em GPU cresceu quando CUDA e StreamSDK foram lançadas em 2006, as quais são interfaces de programação e linguagens projetadas pelos fabricantes de GPUs NVIDIA e AMD/ATI respectivamente.

As próximas seções apresentam um histórico acerca da utilização das GPUs, bem como discorrem sobre o ambiente CUDA.

2.2.1 Histórico

Os processadores gráficos não são tão novos como se imagina: o primeiro processador gráfico para PC foi introduzido em 1982 pela IBM. Foi nomeado CGA (*Color Graphics Adapter*), sendo considerado a primeira placa de vídeo colorida [16]. Equipada com 16 kbytes de memória de vídeo, era capaz de exibir até 16 cores, porém o processamento (rasterização) era totalmente realizado na CPU. Em 1987, a IBM lançou sua última placa gráfica: VGA (*Video Graphics Adapter*), que pode ser classificada como o predecessor de todas as modernas GPUs.

Pelo fato de existirem diferentes características, funcionalidades e resoluções entre as placas gráficas na época, foi criado o VESA (*Video Eletronics Standard Association*), um organismo internacional fundado por fabricantes de adaptadores de vídeo com o objetivo de produzir padrões, a maioria dos quais relativos ao funcionamento de periféricos de vídeo em computadores compatíveis com o IBM PC.

O termo GPU nasceu aproximadamente em 1999, juntamente com o lançamento do primeiro produto da família GeForce da nVidia (GeForce 256) e a declaração em propaganda de "O primeiro GPU voltado para o uso doméstico". Nessa época, três grandes competidores estavam consolidados no mercado: ATI, nVidia e 3Dfx.

Inicialmente, não era feito nenhum processamento na placa de vídeo e não havia possibilidade de desenvolver programas nelas. Com o aparecimento das GPUs, muitas funcionalidades que antes eram feitas pela CPU passaram a ser realizadas diretamente no *hardware* da GPU. A GeForce 256 se destacou por introduzir o conceito de *pipeline* gráfico, no qual a GPU é responsável pela rasterização, texturização, transformação e iluminação dos polígonos. Possibilitou um avanço considerável no desempenho de jogos e foi o primeiro acelerador gráfico compatível com o padrão Direct3D 7. A empresa 3Dfx, adquirida pela nVidia no final de 2000, criou a tecnologia SLI (*Scalable Link Interface*), na qual duas placas de vídeo podem ser interligadas permitindo o processamento paralelo em GPUs. Com a compra da 3Dfx, a disputa pelo mercado concentrou-se entre as empresas nVidia e ATI, e continua assim até os dias de hoje.

Apesar do avanço nas placas de vídeo, somente com o lançamento da GeForce 3 que a GPU tornou-se programável e não somente configurável. Porém, não havia uma linguagem de programação específica, dificultando o desenvolvimento nesses equipamentos. Portanto, para aumentar a flexibilidade do *hardware* dos processadores das placas de vídeo (pois estes

não se adaptavam facilmente à evolução de novos algoritmos e programas de processamento de mídia) [17], surgiram os *Stream Processors* programáveis e linguagens para o programador utilizá-los de maneira simples. Assim, foram lançadas as arquiteturas CUDA da nVidia, o StreamSDK da AMD/ATI e o OpenCL, compatível com ambas tecnologias. A partir de então, a computação de alto desempenho em GPU começou a ganhar destaque e interesse da comunidade científica e do mercado comercial.

Os usos feitos atualmente pelos desenvolvedores, cientistas e pesquisadores para as GPUs são amplos: processamento de imagem e vídeo, computação na área de biologia e química, simulação de fluidos, análises de imagens sísmicas, computação gráfica, entre outros.

Já existe na literatura a implementação de alguns algoritmos de geração de árvore de espalhamento mínimo em GPU, tais como versões modificadas de Prim [18] e boruvka [19]. Vineet et al. [19] propuseram um algoritmo recursivo de Boruvka em GPU-CUDA, em que cada vértice escolhe sua aresta de menor custo e dessa junção surgem os "supervértices". O algoritmo utiliza primitivas básicas, tais como *segmented scan* [20] para encontrar a aresta mínima de cada vértice, e as primitivas *split* e *scan* [21] para juntar vértices e supervértices. Nobari et al. [18] propuseram um algoritmo que aproveita o algoritmo de Prim de modo paralelo, expandindo concorrentemente diversos subconjuntos da floresta de caminhos ótimos computada.

2.2.2 Compute Unified Device Architecture

Em 2006, a NVidia introduziu o CUDA (*Compute Unified Device Architecture*), uma poderosa tecnologia inicialmente disponível nas suas placas da série GeForce 8, que permitia a criação de algoritmos paralelos para execução em suas Unidades de Processamento Gráfico (GPUs). Foi um avanço da empresa no conceito de GPGPU e na computação de alto desempenho. Através dele, a GPU é vista como um conjunto de multiprocessadores capaz de executar um grande número de *threads* em paralelo [16].

Esse modelo de arquitetura e programação possui uma pilha de *software* (Figura 2.1) que envolve uma API com suporte direto a diversas funções matemáticas, primitivas de computação gráfica, bibliotecas, suporte ao *runtime* e ao driver, que otimiza e gerencia a utilização de recursos diretamente com a GPU, esconde a complexidade da placa e permite que os programadores tirem proveito de seu poder de processamento sem a necessidade de conhecer os detalhes do *hardware*.

Diferentemente da CPU, a GPU utiliza a maior parte de seus transistores para cálculos, restando poucos para a parte de controle e cache (Figura 2.2), aumentando o poder de computação e tornando o fluxo do programa mais limitado. Com essa arquitetura, um mesmo trecho de código é executado em paralelo por pequenos blocos de dados, com a existência de várias pequenas caches e diferentes níveis de hierarquia de memória.

A tecnologia CUDA se baseia no *Stream processing* (também conhecido como *Thread Processing*), que é um paradigma de programação de computadores relacionados com a SIMD,

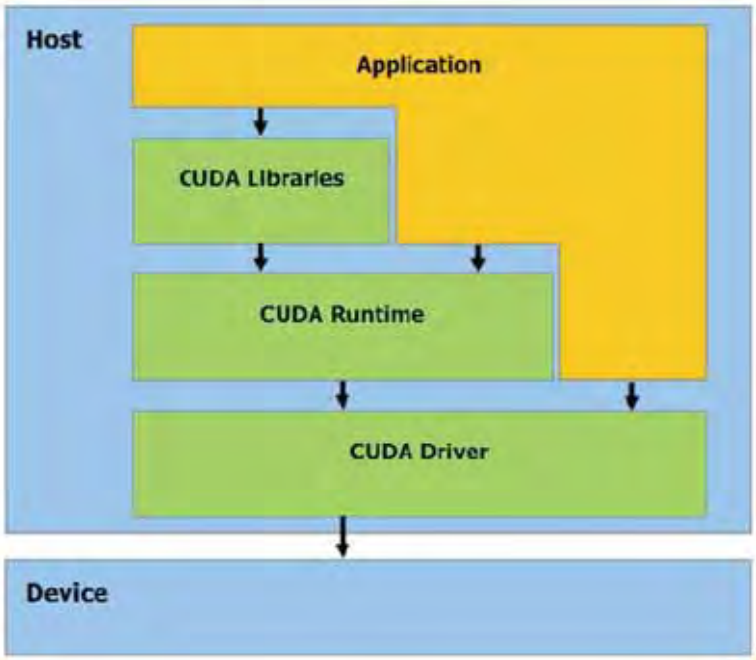


Figura 2.1. Pilha de *software*. Extraído de [1].

Figura 2.2. Alocação de transistores na CPU e GPU. Extraído de [2].

no qual um conjunto uniforme de dados podem ser operados em paralelo e processado por várias instruções: dado um conjunto de dados (*stream*), uma série de operações (funções *kernel*) são aplicadas para cada elemento do *stream*. Este modelo constitui uma forma simples e restrita de paralelismo em que não há necessidade de coordenar a comunicação nem de sincronização explícita.

O princípio básico da programação em CUDA consiste em definir funções *kernels* paralelas, que podem ser funções simples ou programas completos. Um *kernel* executa em paralelo através de um conjunto de N *threads* paralelas que, por serem mais leves que as *threads* que executam em CPU, tem um baixo custo de criação. O programador organiza as *threads* em uma hierarquia de blocos de *threads* e grids de blocos. Cada *thread* no bloco possui um identificador unidimensional, bidimensional ou tridimensional (threadIdx.x, threadIdx.y, threadIdx.z). Um bloco de *threads* é um conjunto de *threads* concorrentes que possuem acesso compartilhado a um espaço de memória privada para o bloco e que cooperam entre si através da barreira de sincronização. Todos os blocos devem ter o mesmo número de *threads* organizadas da mesma forma. Assim como as *threads*, cada bloco de *threads* possui um identificador próprio que pode

conter até três dimensões (blockIdx.x, blockIdx.y, blockIdx.z). O *grid* é um conjunto de blocos de *threads* (Figura 2.3).

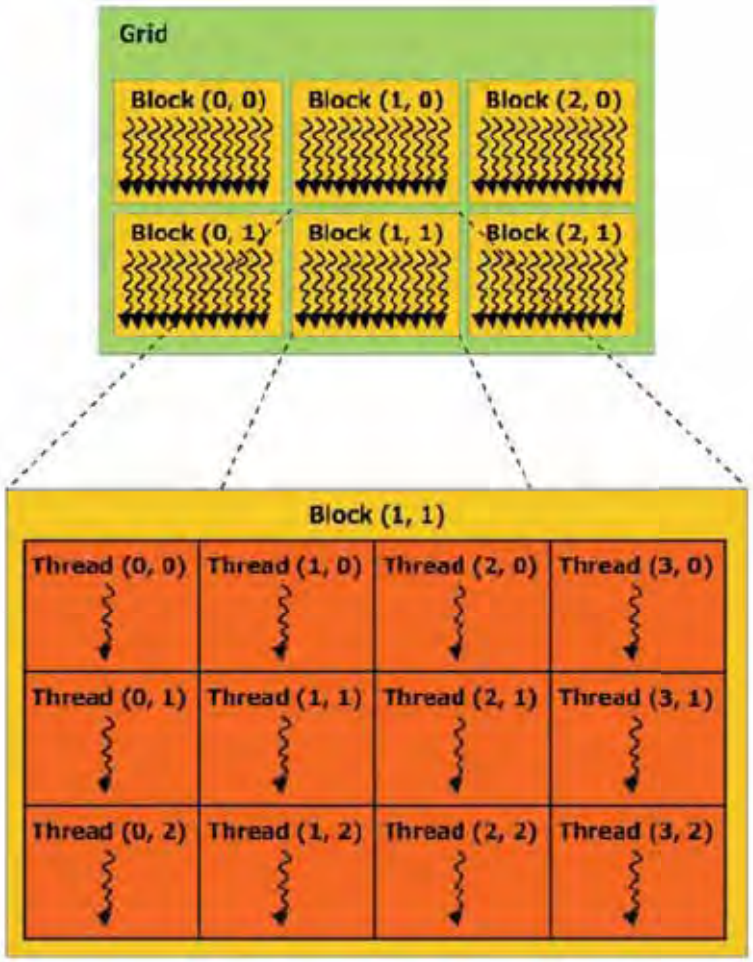


Figura 2.3. *Grid* e bloco de *threads*. Extraído de [2].

A programação em CUDA envolve a execução do código em duas plataformas diferentes ao mesmo tempo: no sistema *host* com uma ou mais CPUs e em um ou mais *devices* com NVIDIA GPU CUDA ativado. As diferenças entre o *host* e o *device* são encontradas principalmente no acesso de memória e *threads*. Enquanto o *host* pode suportar um número limitado de *threads*, o dispositivo suporta no mínimo 768 *threads* ativas concorrentes por multiprocessador. Além disso, a execução de *threads* em CPUs são mais custosas, pois o sistema operacional necessita realizar trocas entre *threads* para manter a capacidade *multithreading*. Já em GPUs, os recursos são alocados para cada *thread* até que esta complete sua execução. Portanto, na arquitetura CUDA a CPU é utilizada para controlar o fluxo de execução do programa enquanto que a GPU é responsável pelos cálculos sobre os dados.

Em relação a memória, existem diferentes tipos (Figura 2.4) com propósitos e necessidades diferentes:

- registrador - memória que cada *thread* possui. É a mais rápida pois fica dentro do multiprocessador, e possui o mesmo tempo de vida que a *thread*.

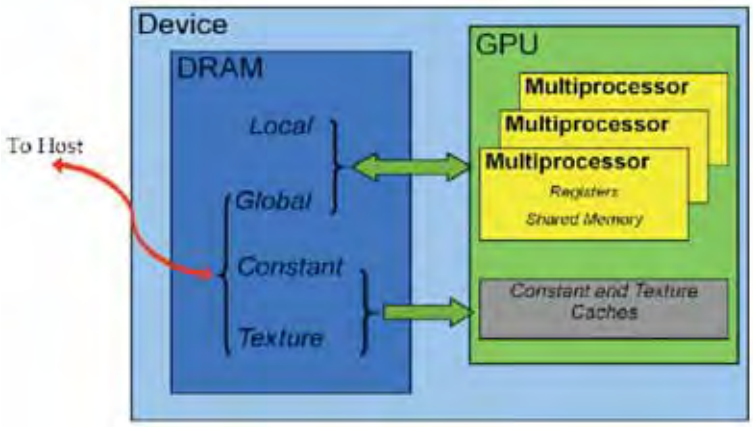


Figura 2.4. Tipos de memória no dispositivo CUDA. Extraído de [3].

- memória compartilhada - memória que cada bloco possui, também é considerada rápida pois é localizada dentro do multiprocessador. É compartilhada entre todas as *threads* que o compõe, possui o mesmo tempo de vida do bloco.
- memória local - desempenho semelhante à memória global, pois se localiza fora do multiprocessador. É uma abstração de uma memória local no escopo de cada *thread*, sendo utilizada quando o conjunto de dados locais ultrapassa o limite da memória mais rápida por alguma razão, possuindo o mesmo tempo de vida da *thread*.
- memória global - todas as *threads* da aplicação tem acesso à mesma memória global, permitindo a comunicação entre blocos de grades diferentes. Localiza-se fora do multiprocessador, e seu tempo de vida é o mesmo da aplicação. É cerca de 150 vezes mais lenta que o registrador e a memória compartilhada.
- textura - memória que pode ser utilizada como alternativa à memória global, obtendo melhor desempenho em determinados casos.
- constante - memória acessível à todas as *threads* e otimizadas para diferentes tipos de utilização, sendo que possui o mesmo tempo de vida da aplicação.

A Figura 2.5 mostra como funciona a hierarquia de memória das *threads*: a memória local de cada *thread*, a memória compartilhada entre as *threads* de um mesmo bloco e a memória global utilizada por todas as *threads* dos blocos de todos os grids.

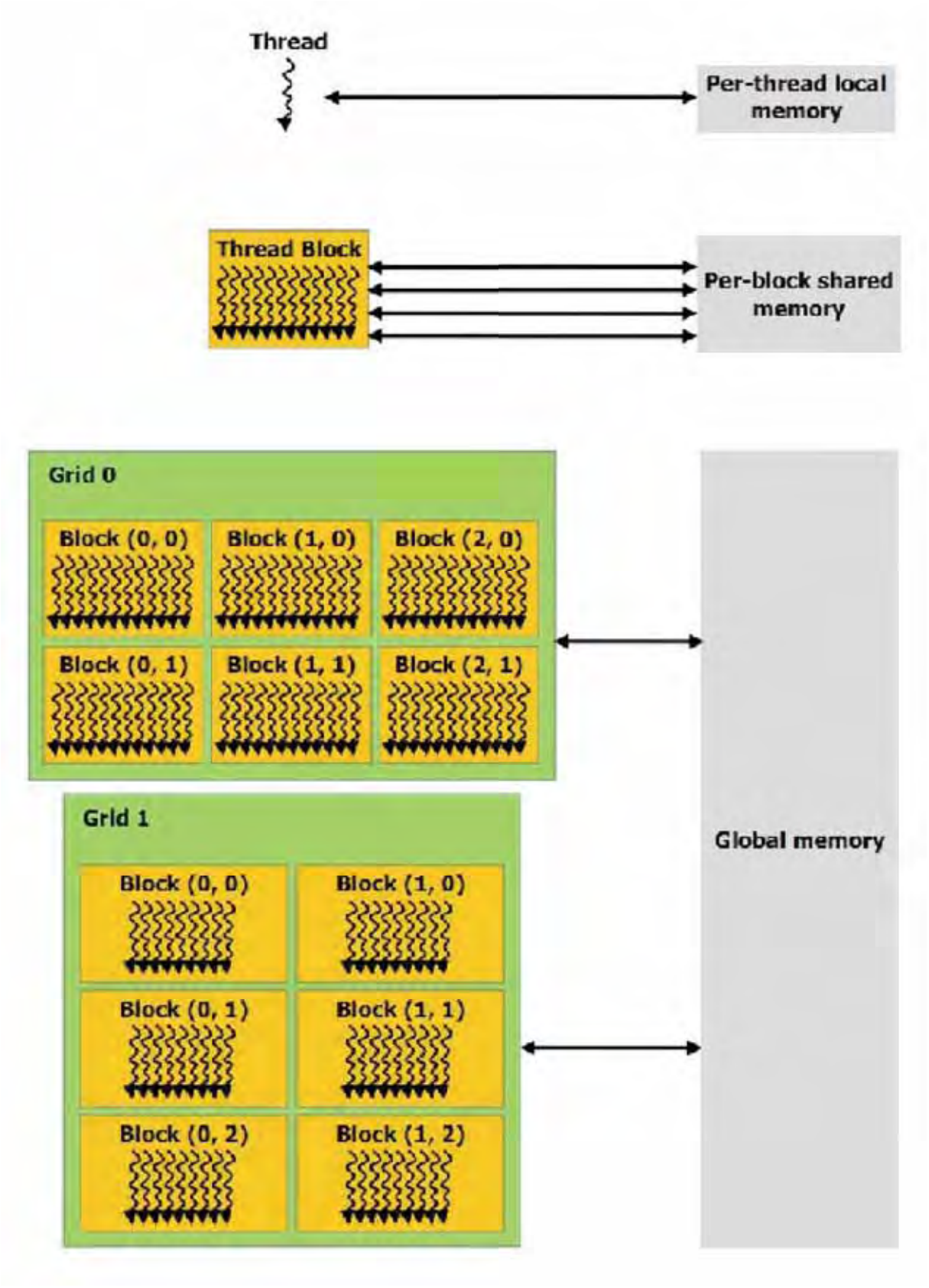


Figura 2.5. Hierarquia de memória. Extraído de [2].

3 Floresta de caminhos ótimos

Este Capítulo tem a finalidade de apresentar os conceitos relacionados ao reconhecimento de padrões por floresta de caminhos ótimos.

A idéia básica de algoritmos de classificação de padrões é: dada uma base de dados, dividir o conjunto em dois tipos: treinamento e teste, sendo que o conjunto de treinamento é utilizado para o aprendizado do comportamento dos dados e o conjunto de teste (ou de classificação) é utilizado para mensurar o aprendizado do classificador. A Figura 3.1 mostra o diagrama desse processo.

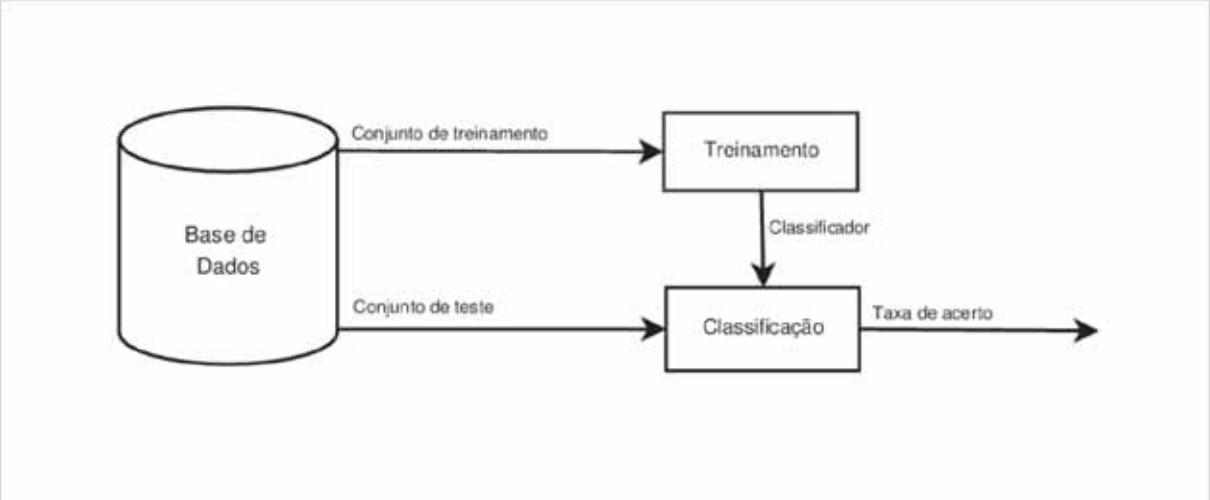


Figura 3.1. Processo de Classificação de padrões

No trabalho em questão, será abordado o reconhecimento por Floresta de Caminhos Ótimos supervisionado, especificamente a fase de treinamento (Figura 3.2), conforme mencionado na introdução.

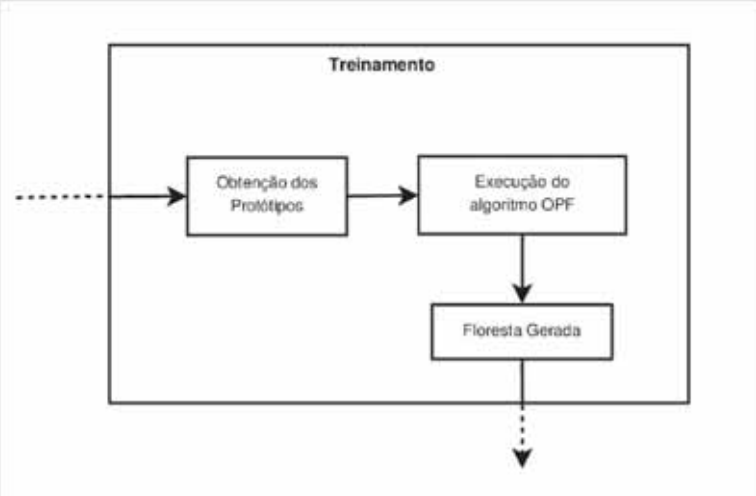


Figura 3.2. Fase de treinamento

3.1 Grafos

Um grafo G é um conjunto $G = (V, E)$ tal que V é o conjunto de n vértices e E o conjunto de arestas conectando pares de vértices.

Um grafo pode ser Direcionado, quando possui arestas representadas por setas e contém Self-loops; ou Não Direcionado quando as arestas (u, v) e (v, u) são consideradas como únicas.

Uma Árvore é um grafo acíclico e conectado, já uma Floresta é um grafo acíclico, podendo ou não ser conectado. A Árvore Geradora T é subgrafo de G tal que $T = (V', E')$, e $E' \leq E, V' = V, T$ é conexo e acíclico.

Uma Árvore de Espalhamento Mínimo (*Minimum Spanning Tree* - MST) tem por objetivo interconectar componentes de um sistema com um menor custo. Dado um grafo conectado não direcionado G, cada aresta $e \in E$ tem peso $w(e)$, que especifica o custo de conexão entre dois vértices. Para resolver o problema de menor custo, deve-se encontrar um subconjunto acíclico $T \subseteq G$ que conecte todos os vértices e cujo peso total $w(T)$ seja minimizado, ou seja, encontrar uma Árvore geradora mínima. Tendo em vista que T é acíclico e conecta todos os vértices, ele deve formar uma árvore, que é chamada de árvore espalhada, pois "se estende" pela amplitude do grafo G.

O algoritmo OPF utiliza a MST para realizar a fase de treinamento, o qual será detalhado na Seção 3.2.1.

3.2 Reconhecimento por Floresta de Caminhos Ótimos

Seja Z uma base de dados e Z_1 e Z_2 os conjuntos de treinamento e teste, respectivamente, com $|Z_1|$ e $|Z_2|$ amostras, as quais podem ser pontos ou elementos de imagem (*pixels*, *voxels*, formas e objetos), tais que $Z = Z_1 \cup Z_2$. Seja $\lambda(s)$ uma função que associa o rótulo correto i, $i = 1, 2, \ldots, c$ da classe i a qualquer amostra $s \in Z_1 \cup Z_2$. Seja, também, $S \in Z_1$ um conjunto de protótipos de todas as classes (isto é, amostras importantes que melhor representam as classes). Seja $\vec{v}$ um algoritmo que extrai n atributos (cor, forma e propriedades de textura) de qualquer amostra $s \in Z_1 \cup Z_2$, e retorna um vetor de atributos $\vec{v}(s) \in \Re^n$. A distância $d(s, t)$ entre duas amostras, s e t, é dada pela distância entre seus vetores de características $\vec{v}(s)$ e $\vec{v}(t)$. Pode-se utilizar qualquer métrica válida (Euclidiana, por exemplo) ou algum algoritmo de distância mais elaborado [22]. O problema consiste em usar S, $(\vec{v}, d)$, Z_1 e Z_2 para projetar um classificador otimal, o qual pode predizer o rótulo correto $\lambda(s)$ de qualquer amostra $s \in Z_2$. Seja (Z_1, A) um grafo completo cujos nós são as amostras em Z_1, onde qualquer par de amostras define um arco em A (isto é, $A = Z_1 X Z_1$) (Figura 3.3a), gerando um grafo completo e ponderado do conjunto Z_1. Note que os arcos não precisam ser armazenados e o grafo não precisa ser explicitamente representado.

Um caminho é uma sequência de amostras $\pi = \langle s_1, s_2, \ldots, s_k \rangle$, onde $(s_i, s_{i+1}) \in A$ para $1 \leq i \leq k - 1$. Um caminho é dito ser trivial se $\pi = \langle s_1 \rangle$. É associado a cada caminho π

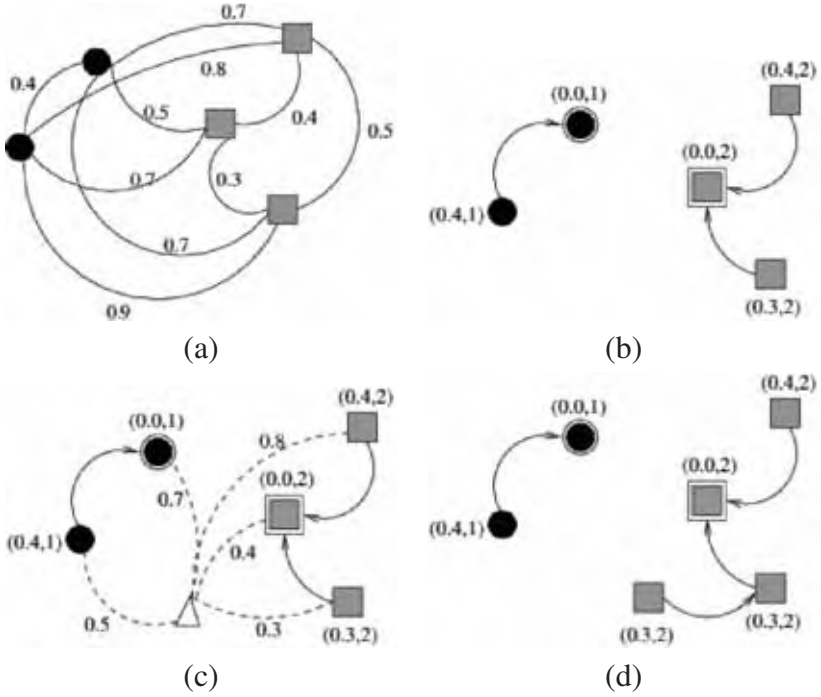


Figura 3.3. (a) Grafo completo ponderado. (b) Floresta de caminhos ótimos resultante para f_{max} e dois protótipos. (c) Amostra de teste e suas conexões com os nós de treinamento. (d) Caminho ótimo do protótipo mais fortemente conexo.

o custo dado por uma função de custos suave f [23], denotada $f(\pi)$. Um caminho π é ótimo se $f(\pi) \leq f(\tau)$ para qualquer caminho τ, onde π e τ terminam no mesma amostra s, independente de sua origem. Também denotamos $\pi \cdot \langle s,t \rangle$ a concatenação do caminho π com término em s e o arco (s,t).

O algoritmo OPF pode ser utilizado com qualquer função de custo suave que pode agrupar amostras com propriedades similares [23]. Entretanto, o OPF foi projetado usando a função de custo f_{max}, por causa de suas propriedades teóricas para estimar protótipos ótimos [24]:

$$\begin{aligned} f_{max}(\langle s \rangle) &= \begin{cases} 0 & \text{se } s \in S, \\ +\infty & \text{caso contrário} \end{cases} \\ f_{max}(\pi \cdot \langle s,t \rangle) &= \max\{f_{max}(\pi), d(s,t)\}, \end{aligned} \tag{3.1}$$

sendo que $f_{max}(\pi)$ computa a distância máxima entre amostras adjacentes em π, quando π não é um caminho trivial.

O algoritmo OPF associa um caminho ótimo $P^*(s)$ de S a toda amostra $s \in Z_1$, formando uma floresta de caminhos ótimos P (uma função sem ciclos, a qual associa a todo $s \in Z_1$ seu predecessor $P(s)$ em $P^*(s)$, ou uma marca nil quando $s \in S$). Seja $R(s) \in S$ a raiz de $P^*(s)$, a qual pode ser alcançada usando $P(s)$. O algoritmo OPF computa, para cada $s \in Z_1$, o custo $V(s)$ de $P^*(s)$, o rótulo $L(s) = \lambda(R(s))$ e o seu predecessor $P(s)$.

A Figura 3.3b mostra a Floresta de caminhos ótimos resultante de Figura 3.3a para f_{max}

e dois protótipos (nós contornados). As entradas (x, y) acima dos nós denotam, respectivamente, o custo e o rótulo das amostras. Os arcos direcionados indicam os nós predecessores no caminho ótimo. Na fase de classificação (Figura 3.3c), é calculado para a amostra de teste (triângulo branco) as suas conexões (linhas tracejadas) com os nós de treinamento. Como resultado da classificação (Figura 3.3d), o rótulo 2 e o custo de classificação 0.3 é associado à amostra de teste, valores obtidos através do caminho ótimo do protótipo mais fortemente conexo.

O algoritmo OPF com grafo completo utilizando f_{max} pode ser entendido como uma transformada de Watershed usando a IFT [25] computada no espaço de características n-dimensional. Além dessa extensão, outras contribuições significativas foram os processos de treinamento e aprendizado, o qual encontra protótipos ótimos (marcadores) na fronteira entre as classes afastando *outliers* (amostras de uma determinada classe que estão presentes em uma região de outra classe no espaço de atributos) do conjunto de treinamento, aumentando a acurácia do classificador.

3.2.1 Treinamento

S^* é um conjunto ótimo de protótipos quando o algoritmo propaga os rótulos $L(s) = \lambda(s)$ para todo $s \in Z_1$. Desta forma, S^* pode ser encontrado explorando a relação teórica entre a Árvore de Espalhamento Mínima (*Minimum Spanning Tree* - MST) [24] e a árvore de caminhos mínimos para f_{max}. O treinamento consiste essencialmente em duas etapas: encontrar S^* e um classificador OPF enraizado em S^*.

Computando uma MST no grafo completo (Z_1, A), obtêm-se um grafo conexo acíclico cujos nós são todas as amostras em Z_1, e os arcos são não direcionados e ponderados (Figura 3.3b). Seus pesos são dados pela distância d entre os vetores de atributos de amostras adjacentes. Esta árvore de espalhamento é ótima no sentido em que a soma dos pesos de seus arcos é mínima se comparada a outras árvores de espalhamento no grafo completo. Na MST, cada par de amostras é conectado por um caminho, o qual é ótimo de acordo com f_{max}. Os protótipos ótimos são os elementos conectados na MST com diferentes rótulos em Z_1, isto é, elementos mais próximos de classes diferentes. Removendo-se os arcos entre classes diferentes, tais amostras adjacentes tornam-se protótipos em S^* e pode-se computar uma floresta de caminhos ótimos sem erros de classificação em Z_1 (processo de competição). Portanto, cada amostra protótipo define sua própria árvore de caminhos ótimos (*Optimum-Path Tree - OPT*), e a coleção de todas as OPTs definem a floresta de caminhos ótimos (OPF), que dá nome ao classificador.

3.2.2 Classificação

A classificação de um elemento t ocorre após a etapa de treinamento, sendo que para qualquer amostra $t \in Z_2$, consideramos todos os arcos que conectam t com as amostras $s \in Z_1$, como se t fizesse parte do grafo de treinamento. Considerando todos os caminhos possíveis de S^* a

t, encontramos o caminho ótimo $P^*(t)$ de S^* e rótulo t com a classe $\lambda(R(t))$ do seu protótipo mais fortemente conectado $R(t) \in S^*$. Este caminho pode ser identificado incrementalmente por meio da avaliação do custo ótimo $C(t)$ como

$$C(t) = \min\{\max\{C(s), d(s,t)\}\},\ \forall s \in Z_1. \tag{3.2}$$

Seja o nó $s^* \in Z_1$ o único que satisfaz a Equação 3.2 (ou seja, o predecessor $P(t)$ no caminho ótimo $P^*(t)$). Dado que $L(s^*) = \lambda(R(t))$, a classificação simplesmente atribui $L(s^*)$ como a classe de t. Um erro ocorre quando $L(s^*) \neq \lambda(t)$.

4 Otimização da etapa de Treinamento

Este capítulo trata dos algoritmos propostos para a etapa de treinamento supervisionado do classificador floresta de caminhos ótimos. Foram desenvolvidas duas versões de algoritmo: otimização em CPU e outra na GPU utilizando o ambiente CUDA.

4.1 Otimização da etapa de Treinamento em CPU

Dizemos que S^* é um conjunto ótimo de protótipos quando o algoritmo OPF minimiza os erros de classificação para cada $s \in Z_1$, sendo que S^* pode ser encontrado explorando a relação teórica entre MST e OPT para f_{max} [24].

Seja G um grafo e F a floresta de espalhamento mínimo (*Minimum Spanning Forest* - MSF) relativa a G, que é essencialmente uma coleção de MSTs. Allene et al. [24] provaram que se F é uma MSF, F é também uma floresta de caminhos ótimos usando a f_{max} como função de custo. Este teorema destaca que, dada uma MST de G, podemos obter a floresta de caminhos mínimos dela, apenas removendo os arcos entre os protótipos e atualizando os custos de todos os nós, o que nós chamamos neste trabalho de **propagação de custo de caminho** [26].

A idéia intuitiva por trás desse teorema diz que os caminhos ótimos dos protótipos para todos os nós irá seguir o formato da MST, e um caminho ótimo de um protótipo de classe branca, por exemplo, precisará passar pelo protótipo de classe preta para alcançar os nós pretos. Como não temos distâncias negativas, um protótipo não poderá conquistar o outro. Portanto, o protótipo preto funciona como uma sentinela que protege os nós pretos de caminhos oriundos dos nós brancos. Deve ser notado que, se existir uma única MST, o erro no treinamento do OPF seria zero.

Assim sendo, este projeto propôs a implementação do algoritmo de treinamento do classificador OPF em ambiente *monothread* na CPU, utilizando a propriedade teórica acima mencionada. A seguir, os algoritmos propostos e implementados pelo presente trabalho: a função principal *opfMSTPrototypeFMax*, que cria a MST armazenando os filhos de cada nó nas estruturas corretas e encontrando os elementos protótipos; e a sua função auxiliar *BuildOpt*, que atualiza os custos de cada nó da árvore enraizada pelo protótipo e corrige a direção das arestas (os predecessores).

As linhas 1 a 5 da função *opfMSTPrototypeFMax* inicializam as variáveis utilizadas no decorrer do algoritmo, sendo que a função $InsereHeap$ insere o nó raiz r ao *heap* Q. O nó r pode ser arbitrário, porém foi implementado utilizando o primeiro nó do grafo. Nas linhas 6 a 21 ocorrem a criação da MST e as descobertas dos nós protótipos. Na linha 6 é realizado um laço onde é verificado se o *heap* Q ainda possui nós. Na linha 7, a função $RemoveHeap$ remove o nó de menor custo do *heap* Q e armazena esse valor em p, enquanto que na linha 8 a variável $pred$ armazena o nó predecessor de p.

Algoritmo 1 – OPFMSTPROTOTYPEFMAX

ENTRADA:	Grafo $G = (V, E)$ conectado, não direcionado e ponderado, tal que V é o conjunto de nós, e cada nó contém uma lista de filhos; e E é o conjunto de arestas $V \times V$, raiz $r \in V$.
SAÍDA:	Floresta de caminhos ótimos T de G.
ESTRUTURAS AUXILIARES:	Heap Q de tamanho $\|V\|$, lista de protótipos $listProt$, estruturas auxiliares q, p, w, tmp e $pred$.

1. **Para** $p = 0$ *até* $|V|$, **faça**
2. $\quad custo[p] \leftarrow \infty$.
3. $custo[r] \leftarrow 0$.
4. $pred[r] \leftarrow NIL$.
5. $InsereHeap(Q, r)$.
6. **Enquanto** $Q \neq \emptyset$, **faça**
7. $\quad p = RemoveHeap(Q)$.
8. $\quad q = pred[p]$.
9. $\quad$ **Se** $q \neq NIL$, **então**
10. $\quad\quad$ **Se** *q e p são protótipos,* **então**
11. $\quad\quad\quad$ *Adicionar q e p em* $listProt$.
12. $\quad\quad$ **Senão**
13. $\quad\quad\quad$ *Insere p na lista de filhos do nó q.*
14. $\quad$ **Para** $w = 0$ *até* $|V|$, **faça**
15. $\quad\quad tmp \leftarrow d(p, w)$.
16. $\quad\quad$ **Se** $tmp < custo[w]$, **então**
17. $\quad\quad\quad pred[w] = p$, $custo[w] \leftarrow tmp$.
18. $\quad\quad\quad AtualizaHeap(Q, w, tmp)$.
19. **Enquanto** $listProt \neq NIL$, **faça**
20. $\quad prot = RemoveSet(listProt)$.
21. $\quad BuildOpt(G, prot)$.

O fluxo de processamento das linhas 9 a 13 só ocorre se $pred$ for diferente de NIL (nulo), sendo que na linha 10 é verificado se $pred$ e p são protótipos (nós conectados de classes diferentes). Em caso positivo, na linha 11 $pred$ e p são adicionados na lista de protótipos $listProt$. Em caso negativo, na linha 13, p é inserido na lista de filhos do nó $pred$. Nas linhas 14 a 18 um laço é realizado com todos os nós de G, sendo que nas linhas 15 a 18 p realiza um processo de conquista de nós: p oferece um custo para w e este é conquistado caso o custo oferecido seja menor do que o custo que ele possui. Se w é conquistado por p, então o predecessor de w é atualizado para p, e o *heap* Q é atualizado com a função $AtualizaHeap$.

Nas linhas 19 a 21 um laço é realizado com todos os elementos da lista de protótipos $listProt$. A variável $prot$ na linha 20 armazena o protótipo removido de $listProt$ utilizando a função de remoção $RemoveSet$. Na linha 21, é realizada uma chamada à função $BuildOpt$,

que tem como parâmetros o grafo G e o protótipo $prot$.

A função $BuildOpt$ basicamente, atualiza os custos e predecessores da árvore de caminhos ótimos enraizada pelo nó protótipo $prot$ passado por parâmetro. Na linha 1, a variável p inicialmente armazena o protótipo $prot$ e o seu custo é setado para 0, e o laço nas linhas 2 a 5 é utilizado para percorrer todos os nós predecessores ao protótipo. Para cada predecessor, é atribuído o custo correto com f_{max} (linha 3), além de ser corrigida a orientação da aresta com o intuito desta apontar para a direção da raiz protótipo. Na linha 4 são adicionados os filhos de p na lista de filhos $listOpt$, a qual contém todos os nós filhos que ainda não foram atualizados. Na linha 5 o novo nó p é atualizado com o seu predecessor.

Nas linhas 6 a 9, o laço percorre todos os nós da $listOpt$ com o intuito de atualizar os custos de todos os nós filhos que ainda não foram atualizados pelo laço anterior. A função $RemoveSet$ da linha 7 remove um nó filho de $listOpt$ e o armazena na variável p. Na linha 8 é atualizado o custo de p com a função f_{max}, onde não é necessário inverter a direção da aresta, pois este já está corretamente direcionado. Os filhos de p são adicionados à $listOpt$ na linha 9 e o fluxo de processamento retorna para a linha 6. A função $BuildOpt$ retorna uma árvore de caminhos ótimos com os custos e arestas devidamente ajustados, tendo o protótipo $prot$ como raiz.

Algoritmo 2 – BUILDOPT

ENTRADA:	Grafo G, protótipo $prot$.
SAÍDA:	Grafo G.
ESTRUTURAS AUXILIARES:	Lista de filhos $listOpt$ e variável q.

1. $p \leftarrow prot$, $custo[0] \leftarrow 0$, $q \leftarrow p$.
2. **Enquanto** $p \neq NIL$, **faça**
3. $\quad custo[p] \leftarrow \max\{custo[q], d(p,q)\}$, *arrumar orientação da aresta.*
4. $\quad$ *Adiciona filhos de p em $listOpt$.*
5. $\quad q \leftarrow p$, $p \leftarrow pred[p]$.
6. **Enquanto** $listOpt \neq NIL$, **faça**
7. $\quad p = RemoveSet(listOpt)$, $q \leftarrow pred[p]$.
8. $\quad custo[p] \leftarrow \max\{custo[q], d(q,p)\}$.
9. $\quad$ *Adiciona filhos de p à $listOpt$.*

A Figura 4.1a ilustra um grafo completo onde as arestas entre os nós denotam a distância entre seus vetores de características correspondentes. O primeiro passo do algoritmo de treinamento do OPF é encontrar os protótipos, o que pode ser feito computando a MST e então marcando os nós mais próximos de classes diferentes. Na Figura 4.1b, os nós do grafo são inicializados com custo ∞ e o nó 1 será a raiz da MST, sendo portanto atribuído a ele o custo zero. A Figura 4.1c ilustra a situação do grafo após três iterações e destaca o nó 4, o qual será o próximo nó a iniciar o processo de conquista dado que os nós 1, 2 e 3 já o fizeram. O nó 4 é o que possui o menor custo dentre os nós que ainda não saíram da fila.

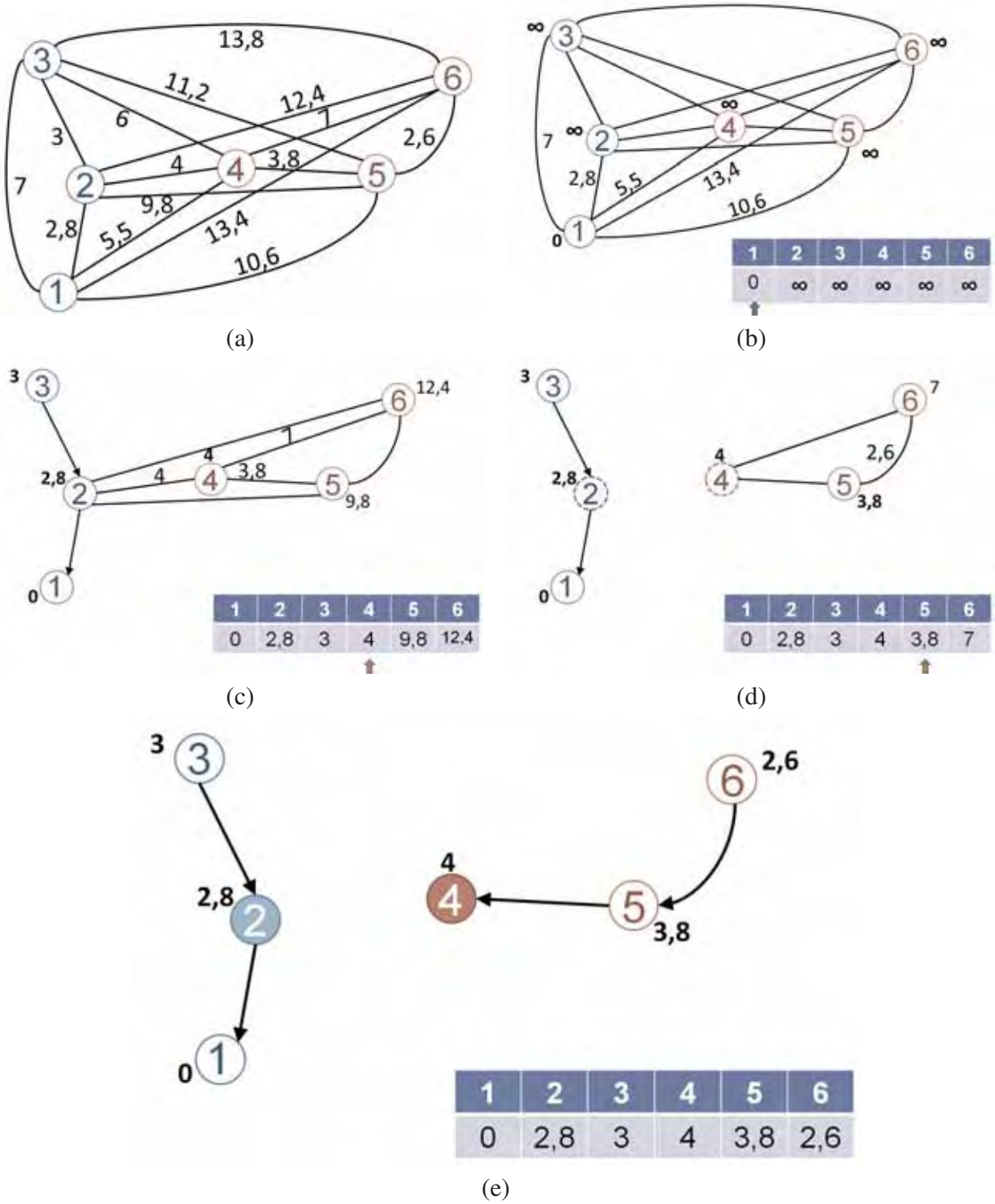


Figura 4.1. (a) Grafo completo ponderado. (b) Inicialização dos nós do grafo. (c) Situação do grafo após três iterações. (d) Após 4ª iteração, dois protótipos são encontrados. (e) Floresta gerada após a primeira fase do treinamento.

Os nós pontilhados na Figura 4.1d representam os protótipos encontrados após a 4ª iteração e, como destacado na tabela, o nó 5 iniciará o processo de conquista. Na abordagem proposta, o arco entre os protótipos é removido, tal como mostra a figura, transformando a MST em uma MSF, enquanto que na abordagem tradicional esse arco é mantido. A Figura 4.1e

apresenta a MSF gerada após a primeira fase do treinamento e os nós destacados definem os protótipos.

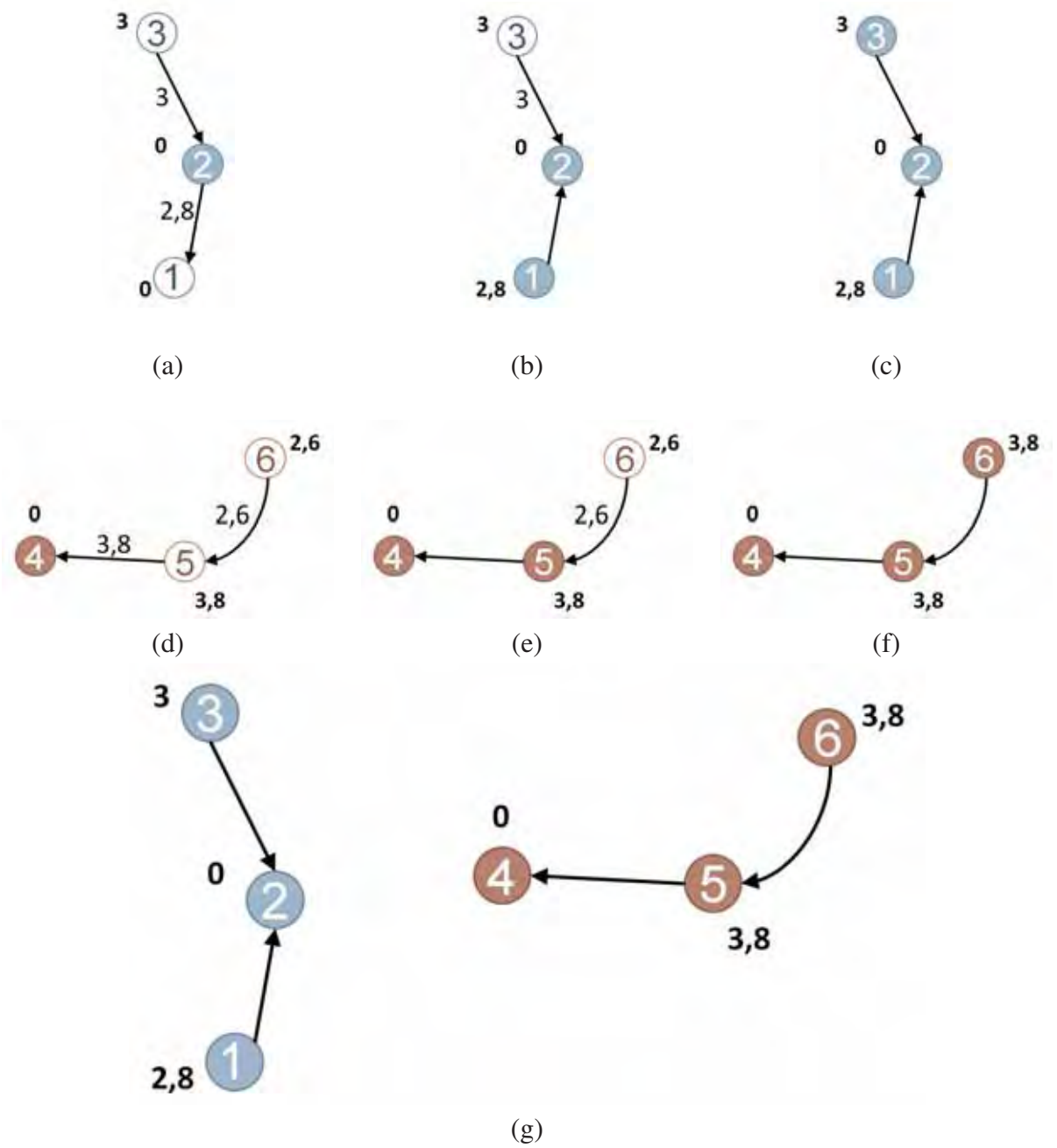


Figura 4.2. (a) Chamada de $BuildOpt$ com nó 2. (b) Atualização de predecessores. (c) Atualização de filhos. (d) Chamada de $BuildOpt$ com nó 4. (e) e (f) Atualização de filhos. (g) Floresta de caminhos ótimos resultante para f_{max} e dois protótipos.

Na abordagem tradicional, após essa primeira fase, executa-se o algoritmo do OPF propriamente dito com o intuito de iniciar o processo de competição entre os nós e então particionar o grafo em OPTs, cada qual enraizadas por um protótipo. Na abordagem proposta, iniciam-se as construções de OPTs com chamadas à função $BuildOpt$ com os protótipos. Assim, podemos obter a floresta de caminhos ótimos apenas removendo a aresta entre os protótipos, direcionando as arestas e conduzindo a propagação de custo do caminho usando f_{max}, não sendo necessário

executar a segunda fase (processo de competição entre os nós), o que torna todo o procedimento de treinamento mais rápido.

A Figura 4.2a ilustra a árvore composta pelos nós 1, 2 e 3 no início do processamento da função $BuildOpt$, sendo o nó 2 o protótipo e raiz da OPT. Como protótipo, é atribuído a ele o custo zero. Já na Figura 4.2b, temos a atualização dos predecessores de 2. Esta etapa é realizada invertendo a aresta (linha 3 do Algoritmo 2) entre os nós (o predecessor de 2 torna-se NIL e o predecessor de 1 é direcionado para 2) e o custo de 1 é determinado por f_{max}. Cabe destacar que, além da inversão da aresta e atualização do custo, é atribuído o rótulo de 2 para 1.

Na Figura 4.2c temos a atualização dos filhos (nós que não precisam atualizar a orientação da aresta, pois já apontam para o caminho que leva à raiz). Nesta fase, é somente atribuído o custo correto ao nó utilizando f_{max} e determinado o seu rótulo como o mesmo de seu pai. A Figura 4.2d ilustra a árvore composta dos nós restantes no início do processamento da função $BuildOpt$. O nó 4 é informado como sendo o nó protótipo e este será a raiz da OPT, sendo atribuído a ele o custo zero. Nas Figuras 4.2e e 4.2f temos a atualização dos filhos. A Figura 4.2g apresenta a floresta de caminhos ótimos resultante.

É importante enfatizar que a abordagem proposta de otimização baseada em propagação do custo do caminho funciona apenas com f_{max}, como indicado em [24].

Experimentos

Esta seção descreve os experimentos realizados para o treinamento em CPU proposto anteriormente. Foram utilizadas diversas bases de dados públicas com diferentes tamanhos para avaliar a eficiência do treinamento e também a acurácia da classificação em cenários distintos, conforme descreve a Tabela 4.1.

Base	Nº Amostras	Nº Características	Nº Classes
Connect [27]	67557	126	3
Covtype (25%) [27]	145250	54	7
IJCNN [28]	49990	22	2
Indian Pine [29, 30]	21025	220	18
Mnist [31]	60000	780	10
Mushrooms [27]	8124	112	2
Salinas [32]	111104	204	17

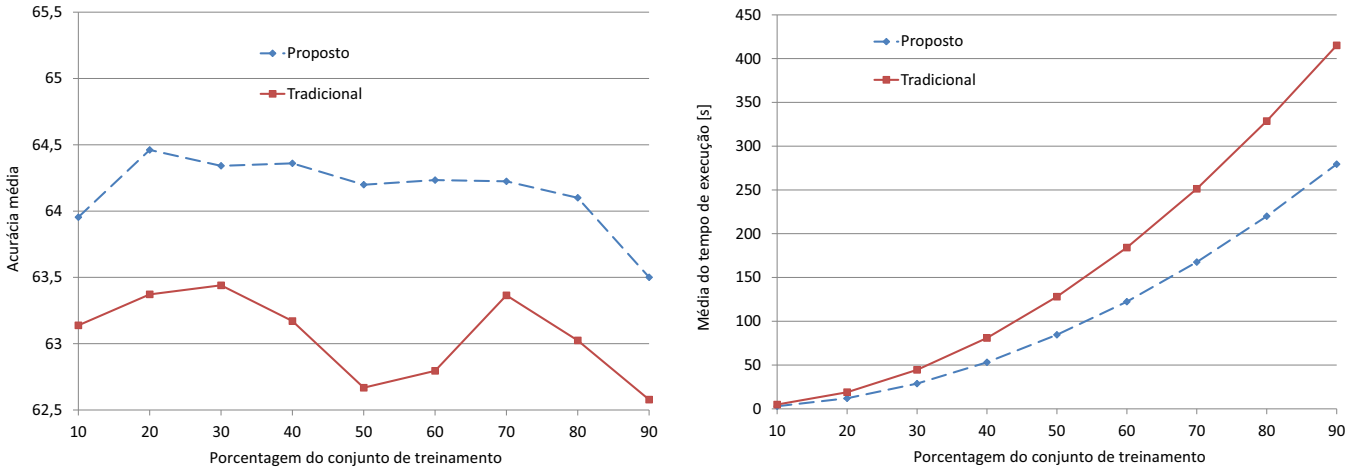
Tabela 4.1. Bases utilizadas nos experimentos.

Com relação aos tamanhos dos conjuntos de treinamento e teste, foi empregado um procedimento de validação cruzada com 10 rodadas com diferentes porcentagens, variando o tamanho do conjunto de treinamento de 10% a 90%, com passos de 10.

O ambiente onde foi realizado os experimentos (tanto em CPU quanto em GPU), em relação ao *hardware*, tem as seguintes características: placa mãe *Intel Corporation DX58SO*

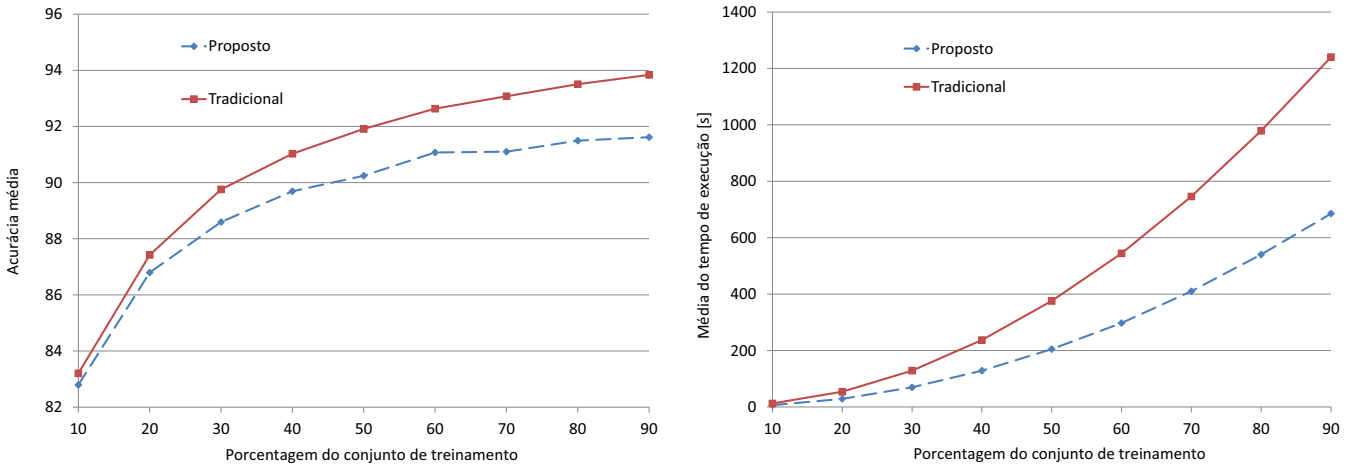
(Smackover); processador *Intel Core i7 - 960 - 3.20GHz* com *8M Cache*; dois pentes de memória *4GB DDR3 1067MHz DIMM Kingston*; disco rígido *Seagate Barracuda 1TB - 7200RPM*; e duas placas de vídeo *NVIDIA Corporation GF 110 (GeForce GTX 570)*. O sistema operacional da máquina era *Ubuntu 12.04 LTS - 64bit* com *Kernel Linux 3.2.0-25*.

As figuras a seguir apresentam os resultados para o algoritmo de treinamento otimizado.



(a) Média da acurácia da classificação (b) Média de tempo de execução (s) da etapa de treinamento

Figura 4.3. Resultados para conjunto de dados Connect.

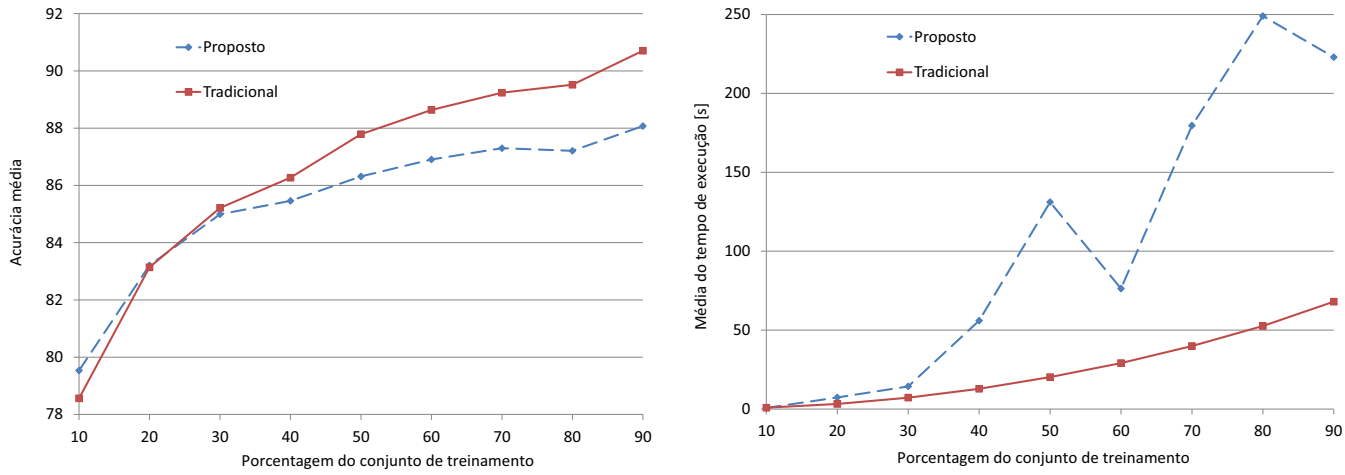


(a) Média da acurácia da classificação (b) Média de tempo de execução (s) da etapa de treinamento

Figura 4.4. Resultados para 25% do conjunto de dados Covtype.

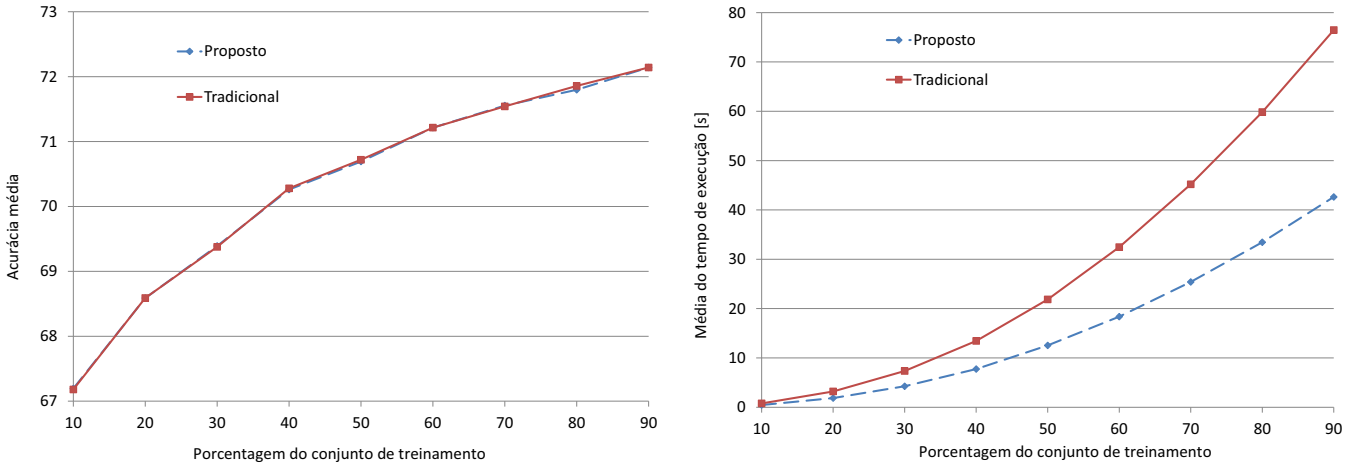
As Figuras 4.3(a) e 4.3(b) apresentam, respectivamente, a média da acurácia e tempo de treinamento para a base de dados Connect. A abordagem proposta teve um ganho de 1,535 para o treinamento (em média). As Figuras 4.4(a) e 4.4(b) apresentam, respectivamente, a média da acurácia e tempo de treinamento para as diferentes porcentagens do conjunto de treinamento e teste para 25% da base de dados Covtype. A abordagem proposta foi 1,842 vezes mais rápido para o treinamento (em média). As Figuras 4.5(a) e 4.5(b) apresentam, respectivamente, a média

da acurácia e tempo de treinamento para a base de dados IJCNN. A abordagem proposta teve um ganho de 0,407 para o treinamento (em média). As Figuras 4.6(a) e 4.6(b) apresentam, respectivamente, a média da acurácia e tempo de treinamento para a base de dados Indian Pine. A abordagem proposta teve um ganho de 1,753 para o treinamento (em média). As Figuras 4.7(a) e 4.7(b) apresentam, respectivamente, a média da acurácia e tempo de treinamento para a base de dados Mnist. A abordagem proposta teve um ganho de 1,960 para o treinamento (em média). As Figuras 4.8(a) e 4.8(b) apresentam, respectivamente, a média da acurácia e tempo de treinamento para a base de dados Mushrooms. A abordagem proposta teve um ganho de 1,777 para o treinamento (em média). As Figuras 4.9(a) e 4.9(b) apresentam, respectivamente, a média da acurácia e tempo de treinamento para a base de dados Salinas. A abordagem proposta teve um ganho de 1,933 para o treinamento (em média).



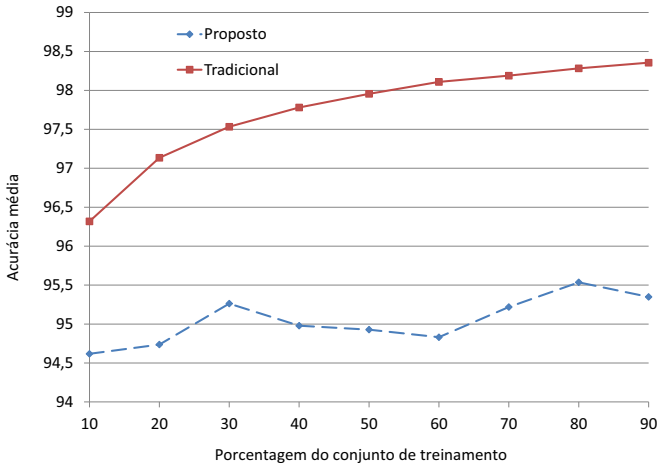
(a) Média da acurácia da classificação (b) Média de tempo de execução (s) da etapa de treinamento

Figura 4.5. Resultados para conjunto de dados IJCNN.

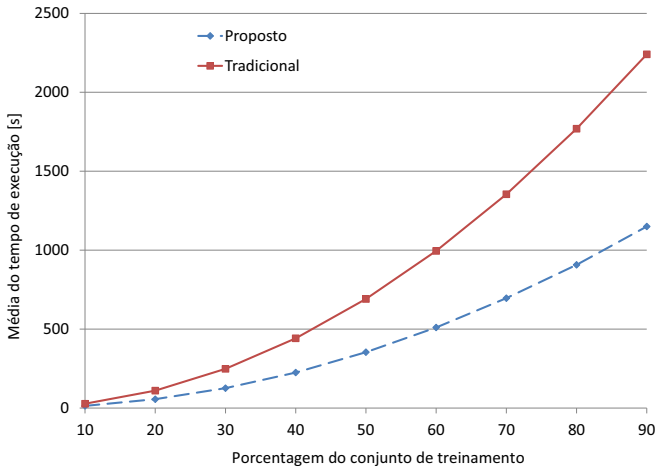


(a) Média da acurácia da classificação (b) Média de tempo de execução (s) da etapa de treinamento

Figura 4.6. Resultados para conjunto de dados Indian Pine.

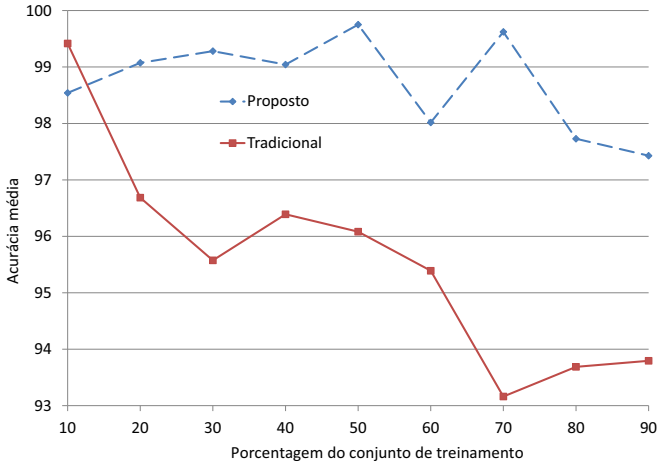


(a) Média da acurácia da classificação

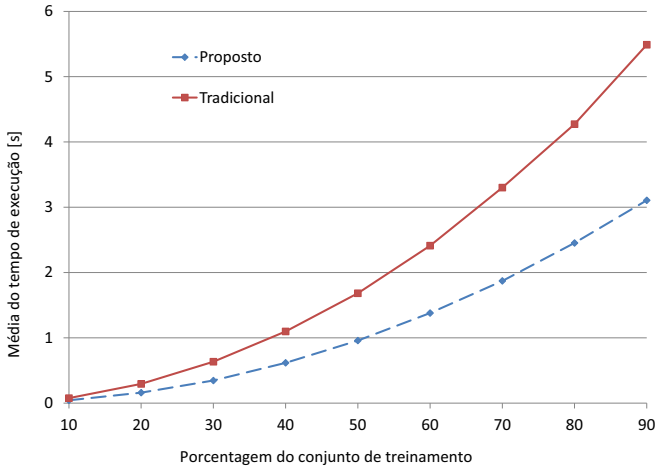


(b) Média de tempo de execução (s) da etapa de treinamento

Figura 4.7. Resultados para conjunto de dados Mnist.

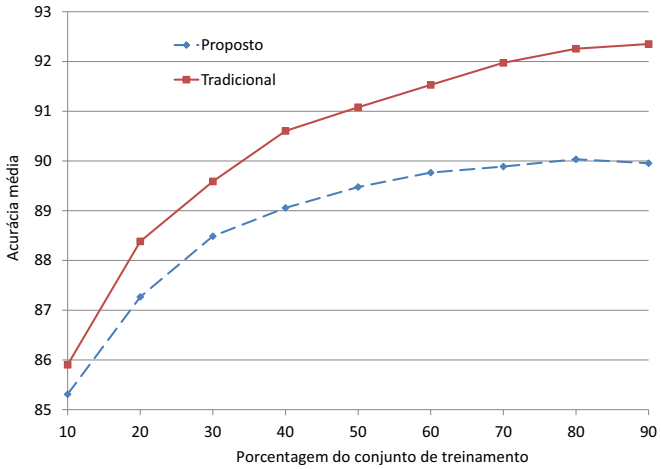


(a) Média da acurácia da classificação

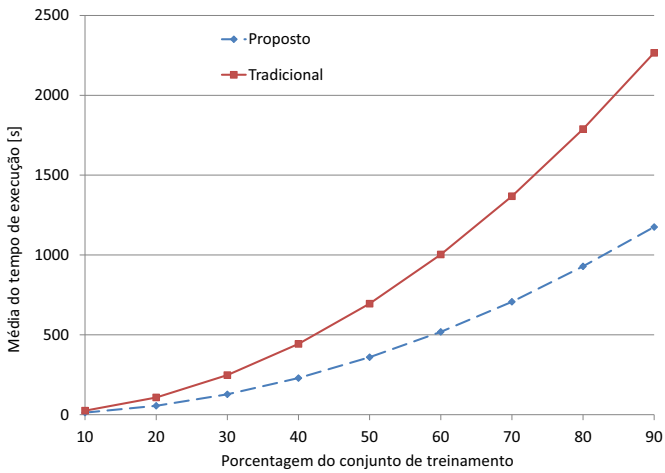


(b) Média de tempo de execução (s) da etapa de treinamento

Figura 4.8. Resultados para conjunto de dados Mushrooms.



(a) Média da acurácia da classificação



(b) Média de tempo de execução (s) da etapa de treinamento

Figura 4.9. Resultados para conjunto de dados Salinas.

Para a base de dados IJCNN verificou-se que a média de tempo de treinamento do algoritmo proposto não foi melhor que o tradicional, porém em todas as outras bases o resultado para o algoritmo proposto foi melhor no tempo de treinamento. Já em relação a acurácia, para a base Mnist o algoritmo tradicional se saiu melhor que o proposto. Em contrapartida, para a base de dados Mushrooms o algoritmo proposto obteve melhor resultados para a acurácia.

É importante salientar que tais diferenças na acurácia são devidas à presença de várias MSTs, isto é, vários caminhos ótimos. Suponha que temos um nó s da classe branca que oferece um caminho de custo ótimo $C_s(t)$ a uma amostra preta t. Agora, suponha que temos um nó preto p que também oferece um caminho de custo ótimo $C_p(t)$, ou seja, $C_s(t) = C_p(t)$. Neste caso, se s saiu da fila antes de p, s conquistará t, e teremos um erro de classificação.

4.2 Otimização da etapa de Treinamento em CUDA

A implementação do algoritmo de treinamento do OPF no ambiente CUDA foi realizado em determinadas instruções do código para que executassem ações semelhantes em paralelo. Como as placas gráficas têm sua estrutura de processamento organizadas em arquiteturas paralelas, os algoritmos nelas implementados devem ter comportamentos paralelizáveis para que ocorra ganho de desempenho.

O código foi paralelizando tanto na parte da criação da MST para encontrar os protótipos, quanto para criar a floresta de caminhos ótimos. Para ambas abordagens, são realizadas três etapas principais: encontrar o elemento p de menor custo que irá realizar a conquista dos demais nós, calcular a distância entre o nó p e demais nós e verificar se o nó p consegue conquistá-los ou não. Para encontrar o elemento de menor custo do conjunto de nós foi utilizada a primitiva *min_element* da biblioteca *Thrust* [33]. Thrust é uma biblioteca de algoritmos paralelos baseado na biblioteca em C++ STL (*Standard Template Library*) e funciona como uma interface de alto nível para aumentar a produtividade dos desenvolvedores.

A seguir, os algorimos propostos e implementados pelo presente trabalho: *cudaopf_Main*, que realiza a alocação e inicialização de variáveis e as chamadas das demais funções; *cudaopf_Treinamento*, responsável por chamar a primitiva paralela *min_element* e a função auxiliar *cudaopf_DistanciaGPU*; a função auxiliar realizada na GPU *cudaopf_DistanciaGPU*, que calcula a distância de um nó para todos os demais nós; e a função realizada na GPU *cudaopf_Prototipos*, que atualiza os custos de cada nó paralelamente. O Algoritmo 3 descreve a função principal que é responsável por inicializar as estruturas do algoritmo e pela chamada das funções.

A função *cudaopf_Main* é o fluxo de instrução principal, responsável por chamar as funções de criar MST para encontrar os protótipos e a função de montar o OPF propriamente dito. Também faz chamada da função paralela de atualização de protótipos e inicializa estruturas na GPU.

Algoritmo 3 – CUDAOPF_MAIN

ENTRADA:	Grafo sg com V vértices alocado na CPU
SAÍDA:	Floresta de caminhos ótimos T de sg.
ESTRUTURAS AUXILIARES:	Vetor $custo$ de melhor custo, vetor de custo $custoAux$ auxiliar e vetor $predecessor$ com os predecessores de cada nó, ambos alocados na GPU.

1. *Aloca $deviceSg$ na GPU usando os dados de sg.*
2. *Aloca matriz $featuresMatrix$ e vetores $custo$, $custoAux$ $isPrototype$ e $predecessor$ na GPU.*
3. *Atribui à matriz $featuresMatrix$ os vetores de $sg.features$.*
4. **Para** $p = 0$ *até* $V - 1$, **faça**
5. $\quad custo[p] \leftarrow \infty$.
6. $\quad custoAux[p] \leftarrow 0$.
7. $\quad isPrototype[p] \leftarrow 0$.
8. $\quad pred[p] \leftarrow NIL$.
9. $custo[0] \leftarrow 0$.
10. $cudaopf\_Treinamento(deviceSg, custo, custoAux, predecessor, isPrototype, featuresMatrix, 1)$.
11. $cudaopf\_Prototipos(isPrototype, custo, custoAux, predecessor)$.
12. $cudaopf\_Treinamento(deviceSg, custo, custoAux, predecessor, isPrototype, featuresMatrix, 0)$.
13. *Copia valores da GPU para CPU.*

Na linha 1 é realizada a alocação da estrutura sg na memória da GPU na variável $deviceSg$. Na linha 2 são feitas as alocações na GPU da: matriz de características $featuresMatrix$; vetor $custo$ com o custo de cada nó; vetor $custoAux$ utilizado para auxiliar na lógica do algoritmo; vetor $isPrototype$ com valor 1 caso o nó seja protótipo, ou 0 caso contrário; e do vetor $predecessor$ com os predecessores de cada nó. Na linha 3 atribuimos os vetores de características de $sg.features$ ao vetor de vetores $featuresMatrix$ (criando assim uma matriz). Com essa matriz que executamos o calculo da distância entre os nós.

Nas linhas 4 a 8 um laço executa V (número de nós) vezes instruções para inicializar os valores dos vetores que foram criados na GPU. Na linha 5 atribuímos para todos os elementos do vetor $custo$ o valor ∞, na linha 6 atribuímos a todos os elementos do vetor $custoAux$ o valor 0, na linha 7 atribuímos no vetor $isPrototype$ todos os nós como não protótipo (valor 0) e na linha 8 atribuímos no vetor $predecessor$ todos os predecessores dos nós como NIL.

Na linha 9 temos a inicialização de um nó com o valor 0 para o início da construção da MST, que será o primeiro nó a "sair" da "fila de nós" da função $cudaopf\_Treinamento$. Na linha 10 é realizada a chamada da função $cudaopf\_Treinamento$ com o parâmetro 1 indicando que é a fase de construção da MST para encontrar os protótipos. Na linha seguinte é feita a chamada da função $cudaopf\_Prototipos$ para atualizar os custos dos protótipos com o valor 0 e predecessor NIL e para os demais nós não protótipos inicializar os custos com valor ∞ para a próxima fase do algoritmo de encontrar a floresta de caminhos ótimos. Na linha 12 é realizada

a chamada da função $cudaopf\_Treinamento$, desta vez com o parâmetro 0, indicando a fase de construção da floresta de caminhos ótimos desejada pelo treinamento. Finalmente, na linha 13 são feitas as cópias dos dados que estão na memória da GPU para as estruturas que estão na memória da CPU. O Algoritmo 4 apresenta o algoritmo de treinamento proposto.

Algoritmo 4 – CUDAOPF_TREINAMENTO

ENTRADA: Grafo $deviceSg$ alocado na GPU com V vértices; vetor $custo$ de melhor custo; vetor de custo $custoAux$ auxiliar; vetor $predecessor$ com os predecessores de cada nó; $featuresMatrix$ é a matriz na GPU com os vetores de características e variável $isMST$ com valores de 0 ou 1.

SAÍDA: Conjunto de protótipos ótimos $isPrototype$ de $deviceSg$ indicando se o nó é ou não protótipo.

1. **Para** $i = 0$ *até* $V - 1$, **faça**
2. $\quad p = thrust :: min\_element(custo)$.
3. $\quad cudaopf\_DistanciaGPU(p, custo, predecessor, isPrototype, featuresMatrix, isMST)$.

A função *cudaopf_Treinamento* é responsável por chamar as funções paralelas: encontrar o nó de menor custo com a função $min\_element$ da biblioteca *Thrust* e o cálculo de distâncias com a função $cudaopf\_DistanciaGPU$.

Na linha 1 a 3 é realizado um laço de repetição de V vezes. Na linha 2, utilizando a função $min\_element$ encontramos de modo paralelo o nó de menor custo: na primeira vez que executar o laço para encontrar a MST, o nó escolhido será aquele atribuído com valor 0; nas primeiras vezes que executar para encontrar a OPF serão escolhidos os nós protótipos; nas demais vezes, para ambos, serão escolhidos para realizar a conquista aqueles nós que já foram consquistados e que possuem menor custo. Na linha 3 é realizado a chamada da função paralela $cudaopf\_DistanciaGPU$, em que irá calcular as distancias do nó escolhido no passo anterior para todos os demais nós, e atualizar os custos destes, caso os nós sejam conquistados. O Algoritmo 5 apresenta o algoritmo de cálculo de distâncias proposto.

A função *cudaopf_DistanciaGPU* é executada por diversas *threads* em paralelo, e a cada conjunto de *threads* é relacionado o nó escolhido pela função *cudaopf_Treinamento* e um nó do grafo. As linhas 1 a 6 são executadas somente na fase de contrução da MST. Na linha 2, é verificado se a classe do predecessor do nó escolhido p é de classe diferente do nó. Caso positivo, nas linhas 3 e 4, são atualizados os valores de $isPrototype$ de ambos os nós com o valor 1, ou seja, marcados como protótipo. Caso contrário, na linha 6, marca-se o $isPrototype$ de p com o valor 0, ou seja, marcado como não protótipo. Na linha 7 atualizamos o valor de $custoAux$ do nó p com ∞, para fins de controle.

Algoritmo 5 – CUDAOPF_DISTANCIAGPU

ENTRADA:	Grafo $deviceSg$ com V vértices; nó de menor custo p; $featuresMatrix$ é a matriz com os vetores de características; vetor $custo$ de melhor custo; vetor $predecessor$ com os predecessores de cada nó e variável $isMST$ com valores de 0 ou 1.
SAÍDA:	Vetores $custo$, $predecessor$ e $isPrototype$.
ESTRUTURAS AUXILIARES:	Variavel $result$ calcula distância caso esteja procurando a MST ou custo do nó caso montando o OPF e variável $valor$ calcula a distância.

1. **Se** $isMST = 1$, **então**
2. **Se** *classe de* $predecessor[p]$ *for diferente da classe de* p, **então**
3. $isPrototype[p] \leftarrow 1$.
4. $isPrototype[predecessor[p]] \leftarrow 1$.
5. **Senão**
6. $isPrototype[p] \leftarrow 0$.
7. $custoAux[p] \leftarrow \infty$.
8. *Calcula distancia usando valor em matriz* $featuresMatrix$ *e armazena na variável* $result$.
9. **Se** $isMST = 0$, **então**
10. $result \leftarrow max(result, custo[p])$.
11. **Se** $result < custo[no\_da\_thread]$, **então**
12. **Se** $isMST = 1$ *e* $custoAux[no\_da\_thread] \neq \infty$, **então**
13. $custo[no\_da\_thread] \leftarrow result$.
14. $custoAux[no\_da\_thread] \leftarrow result$.
15. $predecessor[no\_da\_thread] \leftarrow p$.
16. **Senão se** $isMst = 0$ **então**
17. $custo[no\_da\_thread] \leftarrow result$.
18. $custoAux[no\_da\_thread] \leftarrow result$.
19. $predecessor[no\_da\_thread] \leftarrow p$.
20. $deviceSg.label[no\_da\_thread] \leftarrow deviceSg.label[p]$.

Na linha 8, é calculada a distância entre o nó p e o nó que o conjunto de *threads* "se responsabilizam" de tratar (no_da_thread). Esse cálculo é feito de modo semelhante a uma multiplicação de uma matriz por vetor [34]. As linhas da Matriz de Características (Figura 4.10) são os nós do conjunto de treinamento e as colunas da matriz são os elementos do vetor de característica de cada nó. Esta matriz é alocada na memória de textura, podendo ser acessada por todas as *threads*. O vetor de característica do nó p é alocado na memória global, e as linhas deste vetor são os elementos do vetor de características do nó p. Como realizado em [34], uma *thread* é alocada para cada elemento da Matriz de características, onde fará o cálculo da distância entre um elemento da matriz com um elemento do vetor, guardando este resultado numa matriz com resultados parciais. O tamanho do bloco de *threads* é o mesmo que em [34], com blocos de 16x16. Cada bloco tem uma matriz de Resultados Parciais, que estão alocadas na memória

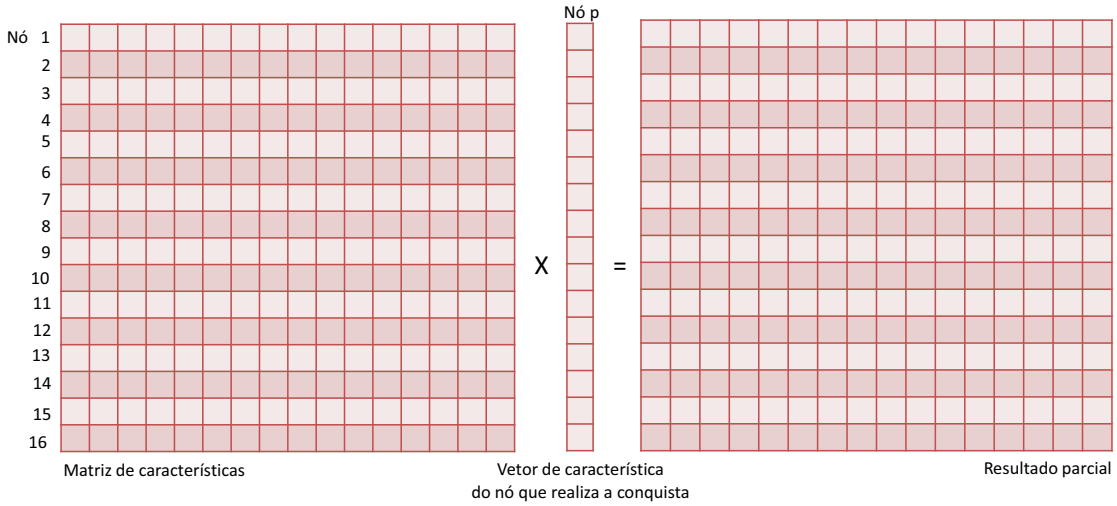


Figura 4.10. Cálculo das distâncias do nó p.

compartilhada de cada bloco de *threads*. O resultado final (distâncias entre p e demais nós) é um vetor de distâncias alocado na memória global, em que é calculado utilizando os valores da matriz de resultados parciais. O nível de paralelismo destas operações pode ser classificado como granularidade fina, já que há um grande número de processamentos pequenos e simples executando em nível de instrução.

As linhas 9 e 10 são executadas apenas na fase de construção da OPF: caso seja a fase do OPF, a função de custo que utilizamos é a f_{max}. Caso seja a fase da MST, a conquista é realizada quando o valor da distância entre p e no_da_thread for menor que o custo de no_da_thread. As linhas 11 a 20 são executadas somente se o valor de $result$ (distância ou custo de p, dependendo do passo anterior) for menor que o custo do no_da_thread. As linhas 12 a 15 são executadas se estiver na fase de construir a MST e se o no_da_thread não tiver "saído" da fila para conquistar os nós do grafo. As linhas 13 e 14 atualizam o $custo$ e o $custoAux$ do no_da_thread com o valor de $result$; ou seja, o no_da_thread é conquistado pelo nó p e é atribuído o valor da distância entre no_da_thread e p. A linha 15 atualiza o predecessor de no_da_thread para o nó p que o conquistou. As linhas 16 a 20 são executadas caso seja esta a fase de construir o OPF. As linhas 17 e 18 atualizam o $custo$ e o $custoAux$ do no_da_thread com o valor de $result$; ou seja, o no_da_thread é conquistado pelo nó p e é atribuído o valor máximo entre a distância entre no_da_thread e p e o custo de p. A linha 19 atualiza o predecessor de no_da_thread para o nó p que o conquistou, e na linha 20 propagamos o rótulo de p para o rótulo de no_da_thread. O Algoritmo 6 apresenta o algoritmo de atualização dos predecessores dos protótipos e dos custos dos nós, que é chamado pela função *cudaopf_Main*.

A função *cudaopf_Prototipos* é executada por diversas *threads* em paralelo, cada uma tratando um nó do grafo. Basicamente, atualiza o custo e o predecessor do nó no_da_thread ao qual é responsável.

As linhas 1 a 4 são executadas se o no_da_thread for protótipo (valor 1). As linhas 2 e

3 atualizam o $custo$ e o $custoAux$ do no_da_thread com o valor 0, pois o custo do protótipo é sempre zero. Já na linha 4 é atualizado o predecessor de no_da_thread com o valor NIL, pois o protótipo é a raiz da árvore e não possui predecessor. Caso o no_da_thread não for protótipo, nas linhas 5 a 7 são atualizados o $custo$ e o $custoAux$ do no_da_thread com o valor ∞, pois estes serão novamente submetidos ao processo de conquista da fase de construção da OPF.

Algoritmo 6 – CUDAOPF_PROTOTIPOS

ENTRADA: Vetor de protótipos $isPrototype$; vetor $custo$ de melhor custo; vetor de custo $custoAux$ auxiliar e vetor $predecessor$ com os predecessores de cada nó.

SAÍDA: Vetores $predecessor$, $custo$ e $custoAux$ atualizados.

1. **Se** $isPrototype[no\_da\_thread] = 1$, **então**
2. $\quad custo[no\_da\_thread] \leftarrow 0$.
3. $\quad custoAux[no\_da\_thread] \leftarrow 0$.
4. $\quad predecessor[no\_da\_thread] \leftarrow NIL$.
5. **Senão**
6. $\quad custo[no\_da\_thread] \leftarrow \infty$.
7. $\quad custoAux[no\_da\_thread] \leftarrow \infty$.

A Figura 4.11a ilustra o conjunto de nós a ser treinado: os nós do grafo são inicializados com custo ∞ e o nó 1 será a raiz da MST, sendo portanto atribuído a ele o custo zero. O primeiro passo do algoritmo de treinamento do OPF é encontrar os protótipos, o que pode ser feito computando a MST e então marcando os nós mais próximos de classes diferentes. A Figura 4.11b ilustra a situação do conjunto de dados quando está na primeira iteração. Com a primitiva *min_element*, o nó 1 sai da fila para a conquista de nós. Cada conjunto de threads na GPU é reponsável por calcular uma distância entre o nó 1 e os demais nós; também verifica se o custo do nó ao qual está "responsável" (*no_thread*) é menor que o valor da distância oferecido pelo nó 1. A Figura 4.11c ilustra a situação do conjunto após a primeira iteração, e a escolha do nó 2 (usando primitiva *min_element*), para o próximo passo. Os nós pontilhados na Figura 4.11d representam os protótipos encontrados após a 4ª iteração e, como destacado pela seta, o nó 4 iniciará o processo de conquista. Do mesmo modo que na Figura 4.11b, o nó 4 sai da fila para a conquista de outros nós, cada conjunto de threads na GPU é reponsável por calcular uma distância entre o nó 4 e os demais nós; também verifica se o custo do nó ao qual está "responsável" (*no_thread*) é menor que o valor da distância oferecido pelo nó 4.

A Figura 4.11e ilustra a situação do conjunto de dados quando está na quinta iteração: a primitiva *min_element* escolhe o nó 5 que sai da fila para realizar a conquista de nós. A Figura 4.11f apresenta a MSF gerada após a primeira fase do treinamento e os nós pontilhados definem os protótipos.

A Figura 4.12a ilustra o conjunto de nós após a construção da MST. Nesta fase, cada *thread* é responsável por atualizar um nó. Cada *thread* verifica se o nó foi atribuído como

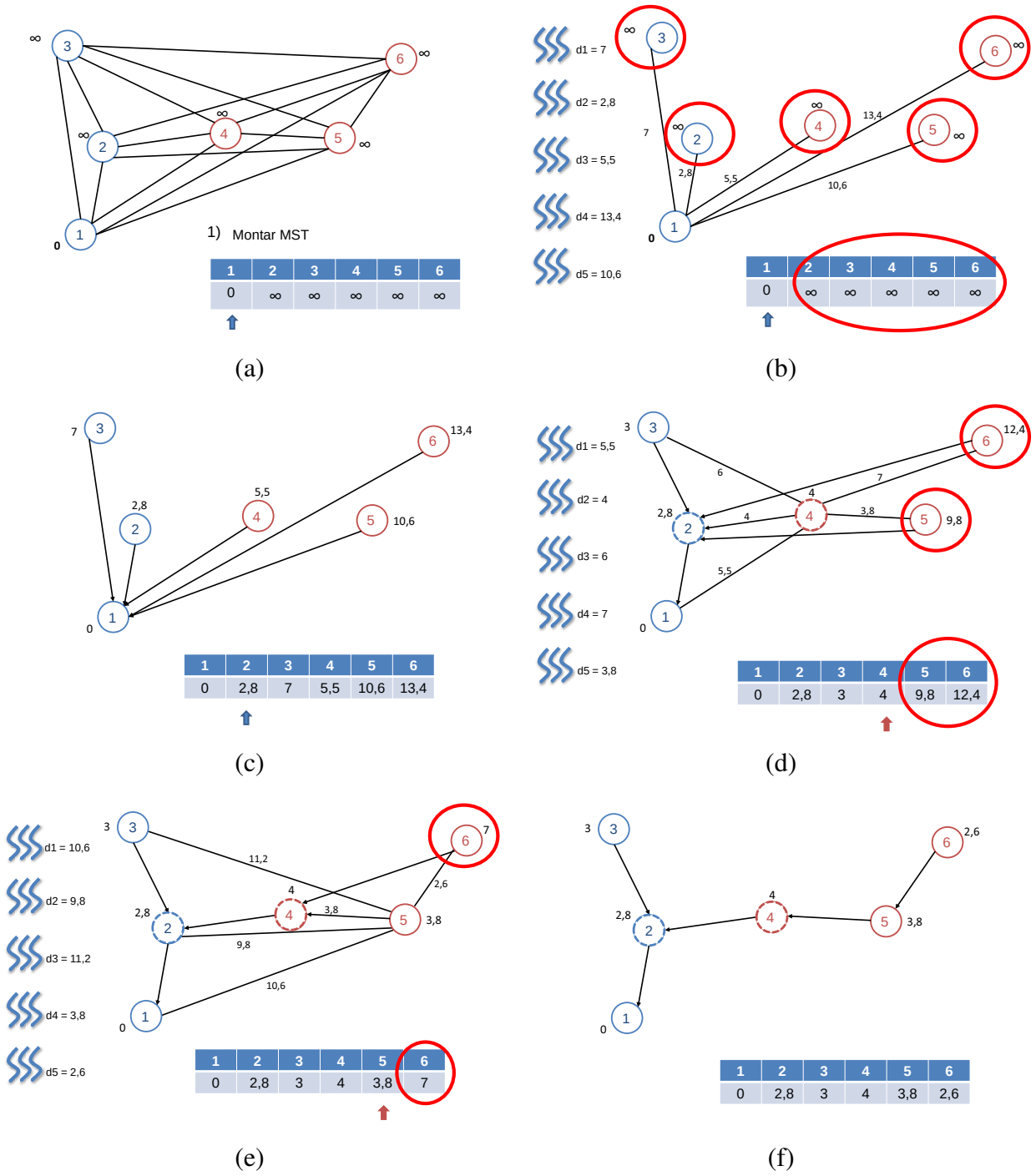
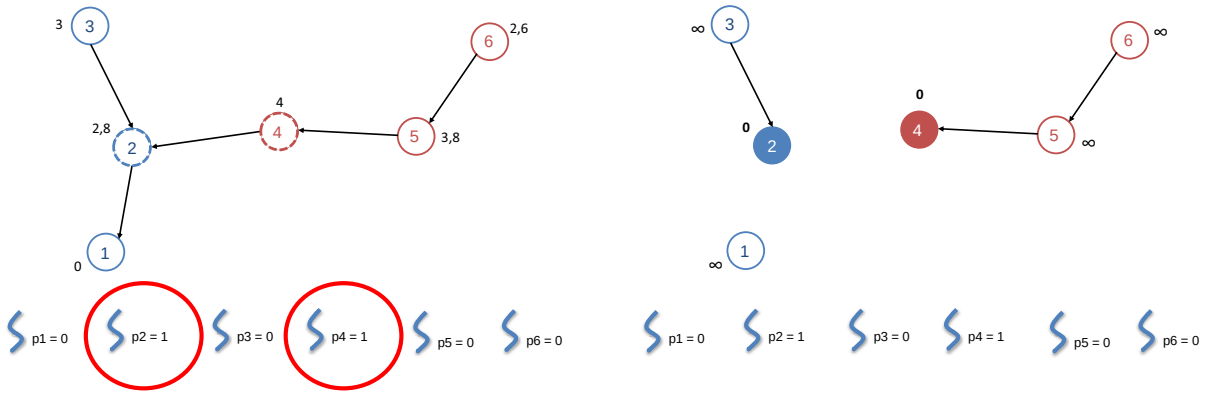


Figura 4.11. (a) Grafo completo. (b) Nó 1 conquista demais nós, cada thread calcula uma distância diferente. (c) Atualiza custos e nó 2 é o próximo a sair da fila. (d) Após 4ª iteração, dois protótipos são encontrados e nó 4 conquista dois nós. (e) Nó 5 sai da fila e conquista nó 6. (f) MST gerada, 2 protótipos encontrados.

protótipo ou não. Caso o nó seja protótipo, a *thread* atualiza o custo do nó com o valor zero e atualiza o predecessor deste nó como NIL. Caso o nó não seja protótipo, o valor do custo será ∞. Já a Figura 4.12b ilustra o conjunto de nós após a atualização dos nós.

A Figura 4.13a ilustra a segunda fase do treinamento: a primitiva *min_element* escolhe o nó protótipo 2 que irá sair da fila para a conquista de nós. A Figura 4.13b mostra a conquista



(a) (b)

Figura 4.12. (a) Cada thread responsável por um nó verifica se é protótipo ou não-protótipo. (b) Cada thread atualiza o custo do nó e, caso este seja protótipo, atualiza também o predecessor.

do nó 2: cada conjunto de *threads* na GPU é reponsável por calcular uma distância entre o nó 2 e os demais nós. Utilizando a função de custo f_{max}, o nó 2 tenta conquistar os demais nós (na primeira iteração, o nó 2 só não conquista os nós protótipos). A Figura 4.13c ilustra a segunda iteração, onde a primitiva *min_element* escolhe o outro nó protótipo 4 que iniciará o processo de conquista. A Figura 4.13d ilustra a quinta iteração, onde a primitiva *min_element* escolhe o nó 5 que iniciará o processo de conquista. A este nó é atribuída a classe de seu predecessor, nó protótipo 4, e os conjuntos de *threads* calculam a distância entre 5 e demais nós, em que realizarão a conquista com a função de custo f_{max}. A Figura 4.13e apresenta a floresta de caminhos ótimos resultante.

Experimentos

Esta seção descreve os experimentos realizados para o treinamento em GPU/CUDA proposto anteriormente. Foram utilizadas diversas bases de dados públicas com diferentes tamanhos para avaliar a eficiência do treinamento e também a acurácia da classificação em cenários distintos, conforme descreve a Tabela 4.2.

Base	Nº Amostras	Nº Características	Nº Classes
Connect [27]	67557	126	3
Covtype (25%) [27]	145250	54	7
IJCNN [28]	49990	22	2
Indian Pine [29, 30]	21025	220	18
Letter [35]	15000	16	26
Mnist [31]	60000	780	10
Poker [27]	25010	10	10
Salinas [32]	111104	204	17

Tabela 4.2. Bases utilizadas nos experimentos.

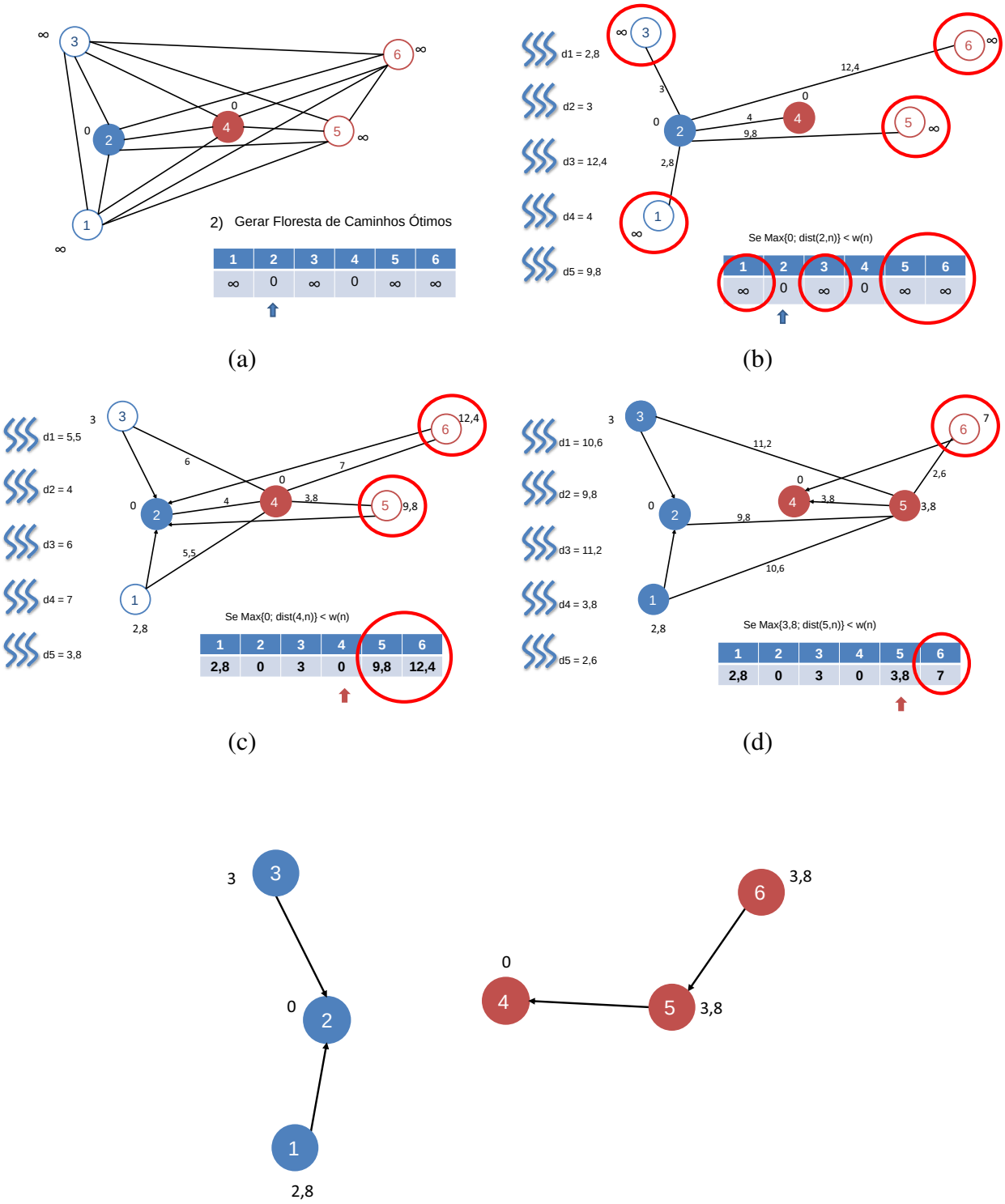
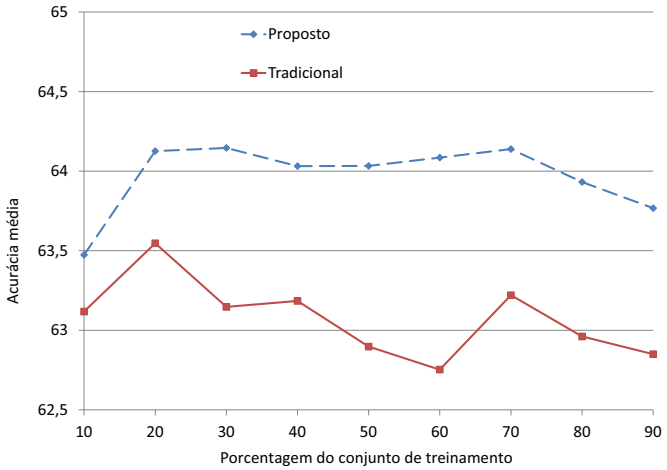
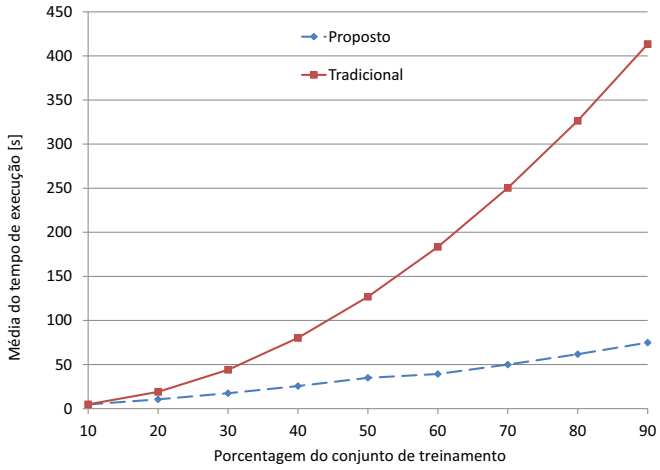


Figura 4.13. (a) Grafo com protótipos encontrados e custos atualizados. (b) Cada thread calcula uma distância, nó protótipo 2 só não conquista o outro protótipo. (c) Na 2ª iteração o nó protótipo 4 conquista dois nós. (d) Na 5ª iteração, nó 5 conquista nó 6 e é atribuído o rótulo do predecessor 4 para o rótulo do nó 5. (e) Floresta de caminhos ótimos resultante.

O ambiente onde foi realizado os experimentos tem o mesmo *hardware* do experimento em CPU; além das seguintes especificações para o trabalho em CUDA: *Cuda compilation tools release 4.2* com 480 *Cuda cores*; e a biblioteca *Thrust* versão *v1.5.1*.

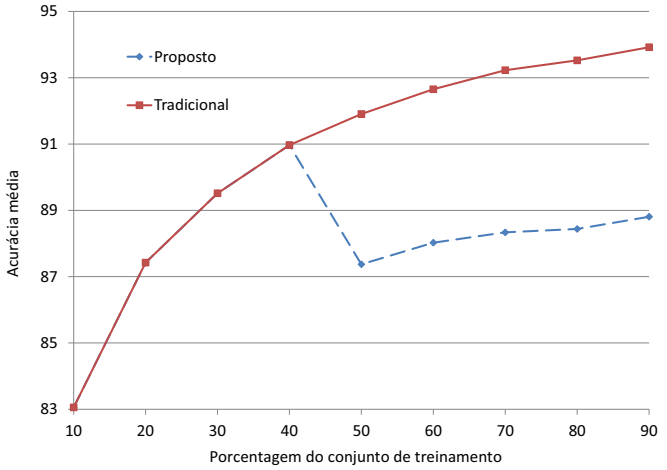


(a) Média da acurácia da classificação

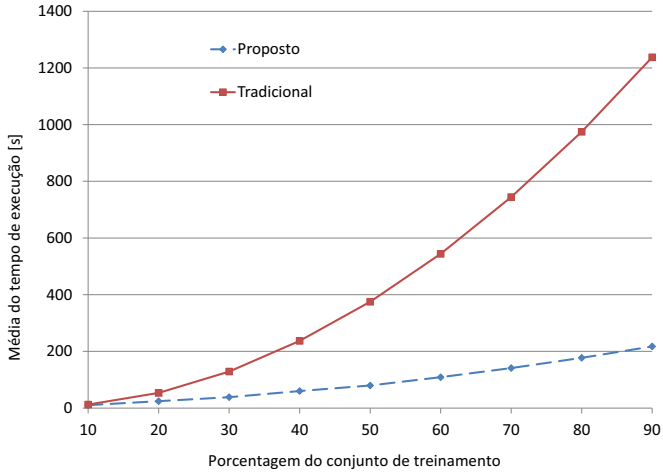


(b) Média de tempo de execução (s) da etapa de treinamento

Figura 4.14. Resultados para conjunto de dados Connect.



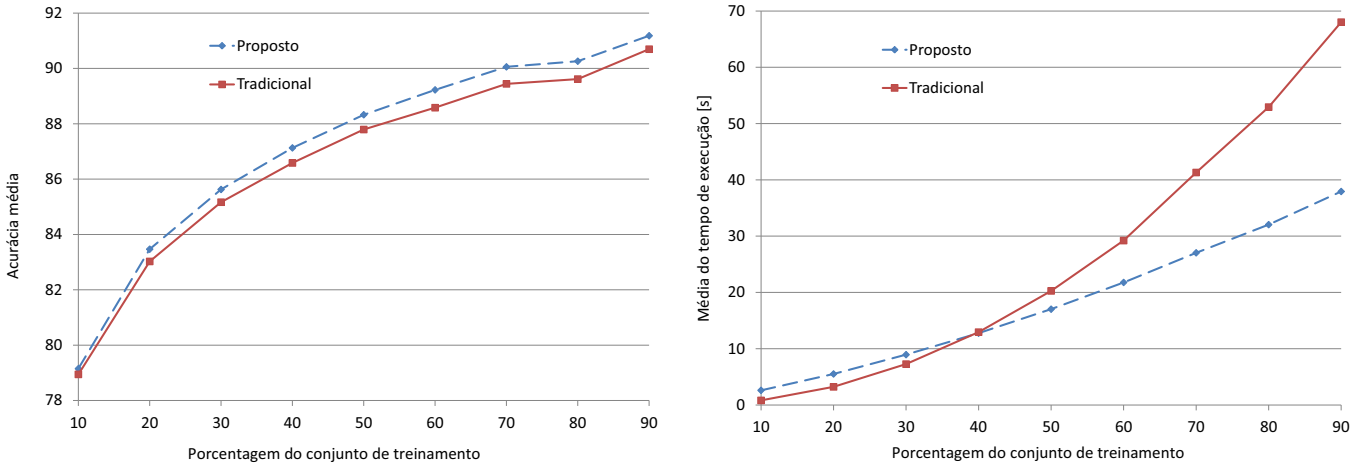
(a) Média da acurácia da classificação



(b) Média de tempo de execução (s) da etapa de treinamento

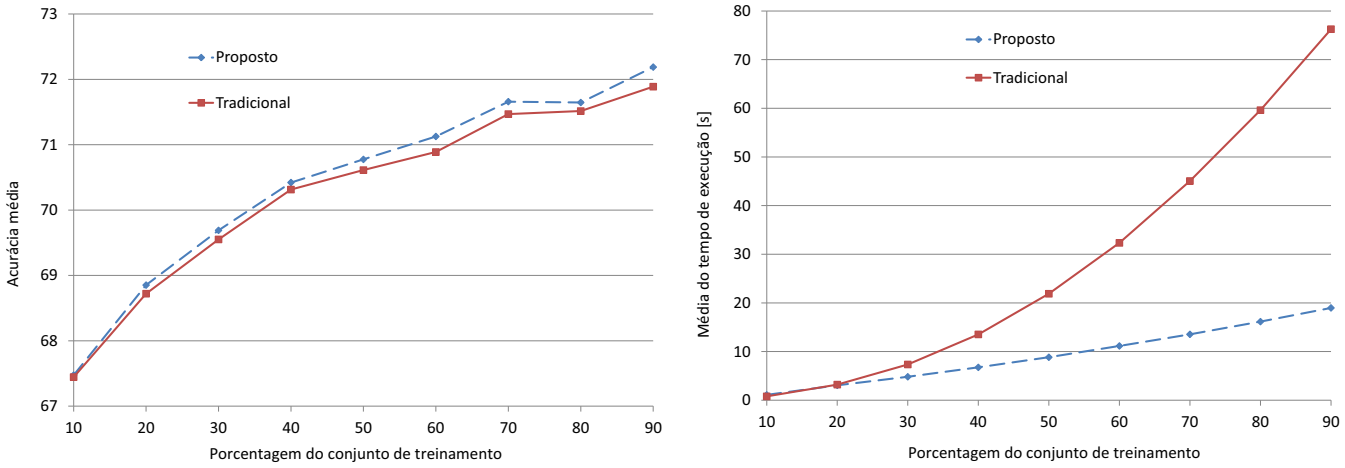
Figura 4.15. Resultados para 25% do conjunto de dados Covtype.

As Figuras 4.14(a) e 4.14(b) apresentam, respectivamente, a média da acurácia e tempo de treinamento para a base de dados Connect. Quando utilizamos o conjunto de treinamento com 30% da base Connect, a abordagem proposta teve um ganho de 2,523 para o treinamento. Já para 90% de base de treinamento, a abordagem proposta foi 5,522 mais rápida. As Figuras 4.15(a) e 4.15(b) apresentam, respectivamente, a média da acurácia e tempo de treinamento para as diferentes porcentagens do conjunto de treinamento e teste para 25% da base de dados Covtype. Quando utilizamos o conjunto de treinamento com 10% da base 25% Covtype, a abordagem proposta teve um ganho de 1,188 para o treinamento. Já para 90% de base de treinamento, a abordagem proposta foi 5,690 mais rápida. As Figuras 4.16(a) e 4.16(b) apresentam,



(a) Média da acurácia da classificação (b) Média de tempo de execução (s) da etapa de treinamento

Figura 4.16. Resultados para conjunto de dados IJCNN.

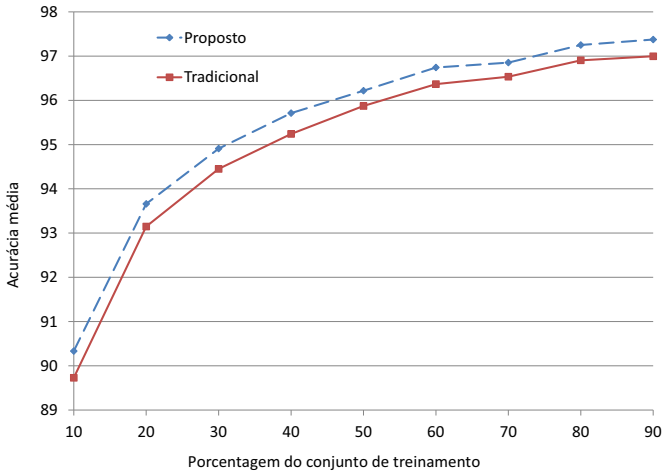


(a) Média da acurácia da classificação (b) Média de tempo de execução (s) da etapa de treinamento

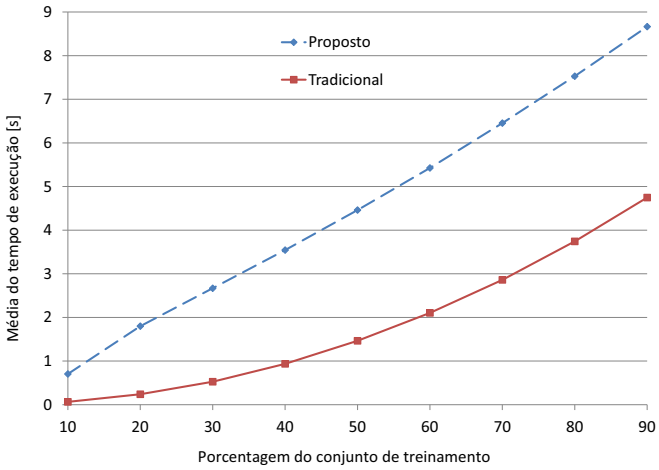
Figura 4.17. Resultados para conjunto de dados Indianpine.

respectivamente, a média da acurácia e tempo de treinamento para a base de dados IJCNN. Quando utilizamos o conjunto de treinamento com 70% da base IJCNN, a abordagem proposta teve um ganho de 1,528 para o treinamento. Já para 90% de base de treinamento, a abordagem proposta foi 1,793 mais rápida.As Figuras 4.17(a) e 4.17(b) apresentam, respectivamente, a média da acurácia e tempo de treinamento para a base de dados Indian Pine. Quando utilizamos o conjunto de treinamento com 40% da base Indian Pine, a abordagem proposta teve um ganho de 2,004 para o treinamento. Já para 90% de base de treinamento, a abordagem proposta foi 4,021 mais rápida.

As Figuras 4.18(a) e 4.18(b) apresentam, respectivamente, a média da acurácia e tempo de treinamento para a base de dados Letter. Verificamos que, quando utilizamos o conjunto de treinamento com 90% de base de treinamento, a abordagem proposta foi 1,825 vezes mais lenta que a abordagem tradicional. As Figuras 4.19(a) e 4.19(b) apresentam, respectivamente,

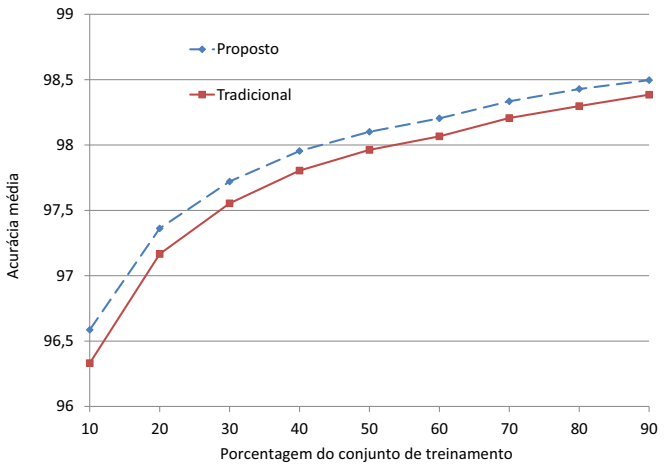


(a) Média da acurácia da classificação

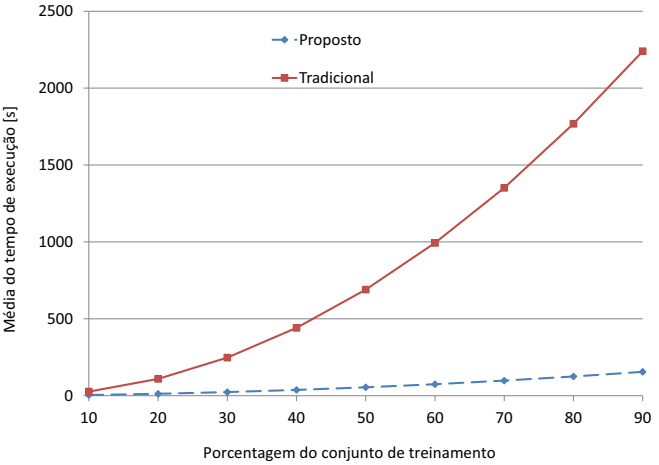


(b) Média de tempo de execução (s) da etapa de treinamento

Figura 4.18. Resultados para conjunto de dados Letter.



(a) Média da acurácia da classificação

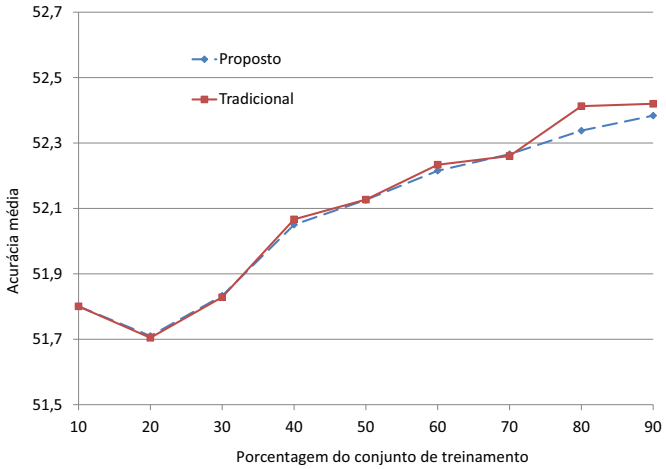


(b) Média de tempo de execução (s) da etapa de treinamento

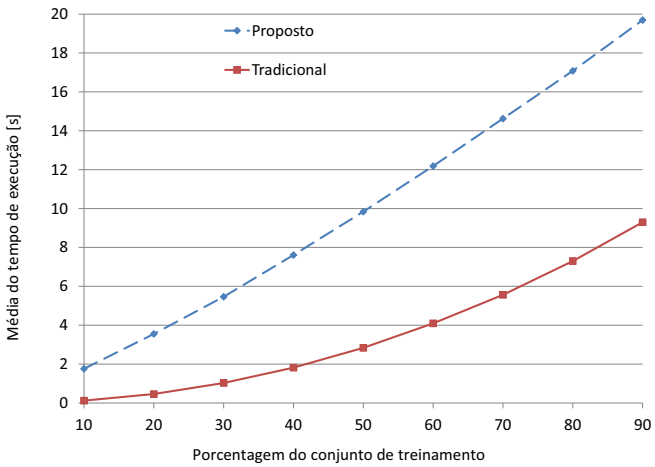
Figura 4.19. Resultados para conjunto de dados Mnist.

a média da acurácia e tempo de treinamento para a base de dados Mnist. Quando utilizamos o conjunto de treinamento com 10% da base Mnist, a abordagem proposta teve um ganho de 5,711 para o treinamento. Já para 90% de base de treinamento, a abordagem proposta foi 14,405 mais rápida.

As Figuras 4.20(a) e 4.20(b) apresentam, respectivamente, a média da acurácia e tempo de treinamento para a base de dados Poker. Verificamos que, quando utilizamos o conjunto de treinamento com 90% de base de treinamento, a abordagem proposta foi 2,118 vezes mais lenta que a abordagem tradicional. As Figuras 4.21(a) e 4.21(b) apresentam, respectivamente, a média da acurácia e tempo de treinamento para a base de dados Salinas. Quando utilizamos o conjunto de treinamento com 10% da base Salinas, a abordagem proposta teve um ganho de 3,038 para o treinamento. Já para 90% de base de treinamento, a abordagem proposta foi 13,580 mais rápida.

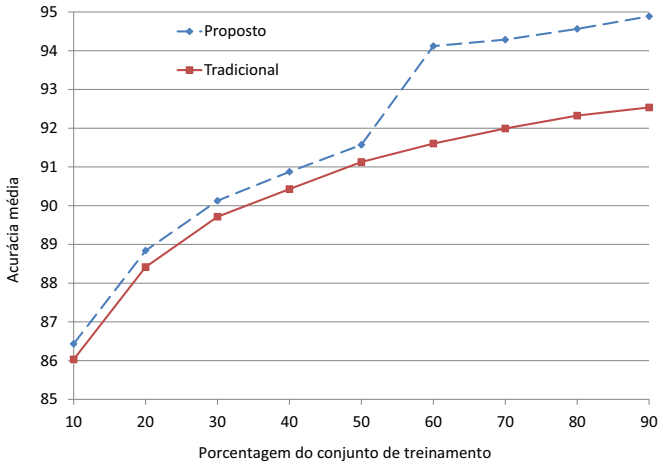


(a) Média da acurácia da classificação

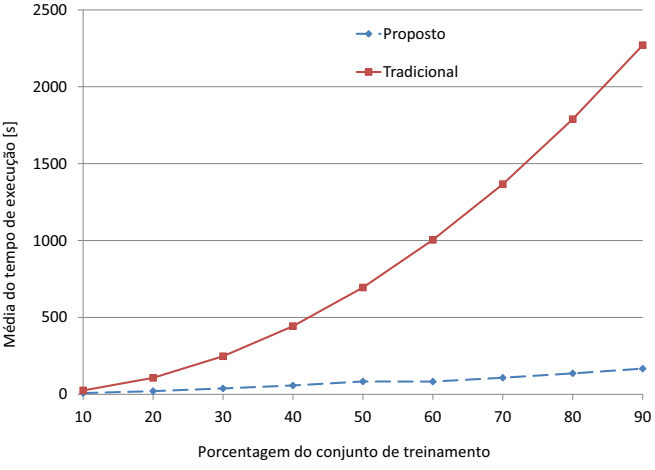


(b) Média de tempo de execução (s) da etapa de treinamento

Figura 4.20. Resultados para conjunto de dados Poker.



(a) Média da acurácia da classificação



(b) Média de tempo de execução (s) da etapa de treinamento

Figura 4.21. Resultados para conjunto de dados Salinas.

5 Conclusões

A área de reconhecimento de padrões é essencial para a computação, podendo também esta ser empregada em diversos domínios de aplicação, tais como medicina, mineração de dados, automação industrial, reconhecimento biométrico e sensoriamento remoto.

Entretanto, em situações nas quais a base de dados é muito grande, o custo do treinamento de um algoritmo de classificação pode não ser satisfatório e demandar muito tempo. Dado que ultimamente empresas de placas de vídeo tem investido cada vez mais em placas gráficas com GPUs, tem-se verificado que está cada vez mais simples e barata a utilização de tais componentes para programação paralela e vários autores já fazem uso desses ambientes.

Neste trabalho, foram estudadas técnicas de programação em ambientes paralelos, bem como o classificador Floresta de Caminhos Ótimos. O presente trabalho focou na possibilidade de reduzir o tempo de treinamento do algoritmo OPF ainda em ambiente CPU quando utilizada a função de custo f_{max}. Assim, um novo algoritmo de treinamento do classificador OPF foi proposto baseado em um estudo teórico que demonstrou que podemos obter uma floresta de caminhos ótimos pela f_{max} apenas direcionando adequadamente as arestas da árvore de espalhamento mínima, removendo os arcos entre os nós protótipos (nós conectados com classes diferentes) e atualizando os custos das amostras. Resultados experimentais demonstraram que a técnica proposta obteve taxas de reconhecimento similares às obtidas pela abordagem tradicional, porém com uma etapa de treinamento mais rápida na maioria dos experimentos realizados.

A etapa de treinamento do algoritmo OPF em ambiente GPU-CUDA foi proposta utilizando a idéia de multiplicação de matriz por vetor de modo paralelo. Resultados experimentais demonstraram que a técnica proposta obteve taxas de reconhecimento similares às obtidas pela abordagem tradicional, porém com uma etapa de treinamento mais rápida na maioria dos casos. Bases que possuem poucos elementos com um vetor de características pequeno não apresentam uma grande melhora no desempenho. Uma possível causa para o não ganho seria o custo que a CPU tem para alocar os dados das *threads* da memória da CPU para a GPU, ou seja, a sobrecarga devido ao carregamento de memória da CPU e GPU. Quando o vetor de características é maior, observamos um ganho de até 14 vezes em determinados casos.

Como trabalhos futuros, um próximo passo é implementar outras versões do classificador OPF em ambiente CUDA, tais com o algoritmo de aprendizado do OPF supervisionado, bem como sua versão não supervisionada.

Esta dissertação obteve um artigo aceito sobre a otimização da etapa de treinamento do classificador OPF em CPU no *International Conference on Pattern Recognition* - ICPR 2012, o qual foi realizado em Tsukuba, Japão (Qualis A1 - Ciência da Computação ano base 2012).

Referências

[1] Nvidia cuda programming guide version 2.0. Available: `http://www.nvidia.com/object/cuda_get.html`, 2008.

[2] Nvidia cuda programming guide version 4.0. Available: `http://www.nvidia.com/object/cuda_get.html`, 2011.

[3] Nvidia cuda c best practices guide version 4.0. Available: `http://www.nvidia.com/object/cuda_get.html`, 2011.

[4] R.O. Duda, P.E. Hart, and D.G. Stork. *Pattern Classification*. Wiley-Interscience, 2 edition, 2000.

[5] S. Haykin. *Neural Networks: a comprehensive fundation*. Prentice Hall, 1994.

[6] C. Cortes and V. Vapnik. Support vector networks. *Machine Learning*, 20:273–297, 1995.

[7] J.P. Papa, A.X. Falcão, and C.T.N. Suzuki. Supervised pattern classification based on optimum-path forest. *International Journal of Imaging Systems and Technology*, 19(2):120–131, 2009.

[8] L.M. Rocha, F.A.M. Cappabianco, and A.X. Falcão. Data clustering as an optimum-path forest problem with applications in image analysis. *International Journal of Imaging Systems and Technology*, 19(2):50–68, 2009.

[9] J. P. Papa and A. X. Falcão. A new variant of the optimum-path forest classifier. In *Proceeding of the 4th International Symposium on Advances in Visual Computing*, pages 935–944, Berlin, Heidelberg, 2008. Springer-Verlag.

[10] B. Catanzaro, N. Sundaram, and K. Keutzer. Fast support vector machine training and classification on graphics processors. In *Proceedings of the 25th international conference on Machine learning*, pages 104–111, New York, NY, USA, 2008. ACM.

[11] T.-N. Do, V.-H. Nguyen, and F. Poulet. Speed up svm algorithm for massive classification tasks. In *Proceedings of the 4th international conference on Advanced Data Mining and Applications*, pages 147–157, Berlin, Heidelberg, 2008. Springer-Verlag.

[12] K.S. Oh and K.C. Jung. Gpu implementation of neural networks. *Pattern Recognition*, 37(6):1311–1314, 2004.

[13] H. Jang, A. Park, and K. Jung. Neural network implementation using cuda and openmp. In *DICTA '08: Proceedings of the 2008 Digital Image Computing: Techniques and Applications*, pages 155–161, Washington, DC, USA, 2008. IEEE Computer Society.

[14] R. Duncan. A survey of parallel computer architectures. *Computer*, 23:5–16, 1990.

[15] W. Hwu D. Kirk. *Programming Massively Parallel Processors: A Hands-on Approach*. Elsevier, 2010.

[16] K. Brodlie R. Borgo. State of the art report on gpus. Technical report, University of Leeds, School of Computing, Visualization and Virtual Reality Research Group, 2009.

[17] R. J. Azevedo B. C. Lopes. High performance computing with cuda. In *Proceedings of the IX Brazilian WSCAD*, 2008.

[18] Sadegh Nobari, Thanh-Tung Cao, Panagiotis Karras, and Stéphane Bressan. Scalable parallel minimum spanning forest computation. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 205–214, New York, NY, USA, 2012. ACM.

[19] Vibhav Vineet, Pawan Harish, Suryakant Patidar, and P. J. Narayanan. Fast minimum spanning tree for large graphs on the gpu. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, pages 167–171, New York, NY, USA, 2009. ACM.

[20] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. Scan primitives for gpu computing. In *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, GH '07, pages 97–106, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association.

[21] Patidar S. and Narayanan P. J. Scalable split and gather primitives for the gpu. Technical report, Centre for Visual Information Technology International Institute of Information Technology Hyderabad - 500 032, INDIA, 2009.

[22] Y. P. Wang and T. Pavlidis. Optimal correspondence of string subsequences. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(11):1080–1087, 1990.

[23] A.X. Falcão, J. Stolfi, and R.A. Lotufo. The image foresting transform: Theory, algorithms, and applications. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(1):19–29, 2004.

[24] C. Allène, J.Y. Audibert, M. Couprie, J. Cousty, and R. Keriven. Some links between min-cuts, optimal spanning forests and watersheds. In *Proceedings of the 8th International Symposium on Mathematical Morphology*, pages 253–264, 2007.

[25] R.A. Lotufo and A.X. Falcão. The ordered queue and the optimality of the watershed approaches. In *Proceedings of the 8th International Symposium on Mathematical Morphology*, volume 18, pages 341–350. Kluwer, 2000.

[26] A. Iwashita, J. P. Papa, A. X. Falcao, R. Lotufo, V. M. Oliveira, V. H. C. Albuquerque, and J. M. R. S. Tavares. Speeding up optimum-path forest training by path-cost propagation. In *21th International Conference on Pattern Recognition*, November 2012.

[27] Covertype. Covertype dataset. uci - machine learning repository. Available: `http://archive.ics.uci.edu/ml/datasets/Covertype`, 2012.

[28] Danil Prokhorov. Ijcnn 2001 neural network competition. Available: `http://www.geocities.com/ijcnn/nnc_ijcnn01.pdf`, 2001.

[29] D. Landgrebe. Indian pines aviris hyperpsectral reflectance data: 92av3c, 1992. A 145×145 pixel and 220 band subset in BIL format of radiance data that has been scaled and offset to digital numbers, DN, from radiance, rad in units $W/(cm^2 * nm * sr)$ according to DN = 500 * rad + 1000 as 2bit signed integers. It is available at `ftp://shay.ecn.purdue.edu/pub/biehl/MultiSpec/92AV3C`.

[30] D.A. Landgrebe. *Signal Theory Methods in Multispectral Remote Sensing*. Wiley, Newark, NJ, 2005.

[31] Yann LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. Available: `http://yann.lecun.com/exdb/mnist/`, November 1988.

[32] S. Kaewpijit, J.L. Moigne, and T.A. El-Ghazawi. Automatic reduction of hyperspectral imagery using wavelet spectral analysis. *IEEE Transactions on Geoscience and Remote Sensing*, 41(4):863–871, 2003.

[33] Jared Hoberock and Nathan Bell. Thrust: A parallel template library. Available: `http://www.meganewtons.com/`, 2010. Version 1.3.0.

[34] N. Fujimoto. Faster matrix-vector multiplication on geforce 8800gtx. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–8, 2008.

[35] R. D. King, C. Feng, and A. Sutherland. Statlog: Comparison of classification algorithms on large real-world problems. *Applied Artificial Intelligence*, 9(3):289–333, 1995.

Autorizo a reprodução xerográfica para fins de pesquisa.

São José do Rio Preto, 18 / 05 / 13

Adriana Sayuri Iwashita
Assinatura