Impulsionando Árvores Extremamente Aleatórias em Paralelo para a Classificação de Dados Textuais

Julio C. B. Pires[1], **Wellington S. Martins**[2]

[1]Instituto Federal de Educação, Ciência e Tecnologia Goiano (IF Goiano)
Campus Avançado Hidrolândia – 75.340-000 – Hidrolândia – GO – Brazil

[2]Instituto de Informática – Universidade Federal de Goiás (UFG)
Campus Samambaia – 74.690-900 – Goiânia – GO – Brazil

julio.pires@ifgoiano.edu.br, wellington@inf.ufg.br

Abstract. *Ensembles of decision trees stand out as a strong competitor for document classification. A drawback though is the high computational cost due to large vocabularies (high dimensionality) and expensive tree building operations. Parallelism exploitation has been an alternative to alleviate these problems. In this work we propose a parallel algorithm to accelerate the construction of these decision trees as part of a recent method that has been shown to surpass state-of-art classifiers for textual data. Experimental results using standard textual databases show that the algorithm implemented on a manycore architecture(GPU) is able to reduce the execution time by up to 26 times compared to an equivalent sequential algorithm.*

Resumo. *Os algoritmos de aprendizado usando conjuntos de árvores de decisão têm se destacado na classificação de documentos, mas não sem pagar um alto custo computacional. A exploração de paralelismo tem sido uma alternativa para viabilizar o uso destes algoritmos mais sofisticados. Neste trabalho propomos um algoritmo paralelo para acelerar a construção destas árvores de decisão utilizadas num método recente que demonstrou superar os classificadores de última geração para dados textuais. Resultados experimentais, utilizando bases de dados textuais padronizadas, mostram que o algoritmo implementado em uma arquitetura manycore (GPU) é capaz de reduzir o tempo de execução em até 26 vezes em comparação a um algoritmo sequencial equivalente.*

1. Introdução

Os avanços recentes em tecnologias da informação e comunicação resultaram em um volume massivo de dados de diferentes fontes e formatos. A quantidade disponível atualmente é medida em Zettabytes (ZB) e é usualmente referida como *Big Data* [Alharthi et al. 2017]. Se de um lado existe muita informação, de outro, surgem grandes desafios para encontrar padrões úteis e importantes escondidos nos dados. Neste sentido está o Aprendizado de Máquina, um método que aprende com os dados, ou seja, usa algumas entradas (variáveis independentes) para descobrir o valor da saída (variável dependente) [Hastie et al. 2009]. Uma subárea do aprendizado é a Classificação, que consiste em categorizar itens de objetos em classes determinadas previamente.

A construção de um classificador pode ser muitas vezes computacionalmente onerosa, e soluções baseadas na execução sequencial estão se tornando cada vez mais

inviáveis [Jansson et al. 2014]. A compra de equipamentos melhores não têm conseguido ganhos expressivos de desempenho quando o paralelismo não é adequadamente explorado em arquiteturas modernas [Navarro et al. 2013]. Assim, o processamento de grandes volumes de dados tem se valido de estratégias paralelas [Chiu et al. 2011], para aumentar seu desempenho. As *GPUs* têm obtido grande sucesso na aceleração de algoritmos de aprendizado de máquina em conjuntos de dados significativamente maiores em muitas áreas de aplicação [Cano 2018].

No caso específico de classificação de documentos, algoritmos cada vez mais sofisticados têm sido utilizados. A alta dimensionalidade e esparsidade da representação de documentos aliada a técnicas de *ensemble* (montagem de conjuntos) tem requerido um grande poder computacional. Os algoritmos de aprendizado baseados em conjuntos de árvores de decisão têm se destacado nesta tarefa, mas não sem pagar um alto custo computacional advindo da alta dimensionalidade dos dados e dos vários modelos (árvores) que precisam ser criados. Assim, a exploração de paralelismo tem sido uma alternativa para viabilizar o uso destes algoritmos em grande bases de dados. Neste trabalho propomos um algoritmo paralelo para acelerar a construção destas árvores de decisão utilizadas num método recente que tem sido capaz de superar os classificadores de última geração para conjunto de dados textuais, o *BERT* [Campos et al. 2017]. Resultados experimentais, utilizando bases de dados textuais padronizadas, mostram que o algoritmo implementado em uma arquitetura manycore (*GPU*) é capaz de reduzir o tempo de execução em até 26 vezes em comparação a um algoritmo sequencial equivalente.

O artigo está organizado da seguinte forma. Na Seção 2 são apresentados conceitos e técnicas de programação para *GPU*. Na Seção 3 é abordado o funcionamento da classificação com árvores/florestas de decisão. A Seção 4 compõe os trabalhos relacionados relevantes para esta pesquisa. A Seção 5 descreve o algoritmo paralelo proposto. Na Seção 6 é feita uma análise e descrição dos resultados obtidos. Na Seção 7 são apresentadas as considerações finais e trabalhos futuros.

2. Unidades de Processamento Gráfico

A arquitetura de uma *GPU* possui dois níveis de paralelismo. No primeiro nível ficam os multi-processadores de fluxo (*SM*), enquanto os processadores de fluxo (*SP*) ficam no segundo nível dentro de cada multi-processador. Assim, um programa paralelo na *GPU* é primeiro dividido em blocos de computação que podem ser executados independentemente nos *SMs*, sem se comunicarem entre si. Esses blocos devem ser divididos em tarefas menores (*threads*) que são executadas nos *SPs*, mas com cada *thread* podendo se comunicar com outras *threads* no mesmo bloco. Cada uma dessas *threads* tem acesso a uma memória global de alta capacidade, bem como a uma memória compartilhada com capacidade reduzida, mas rápida, e os registradores.

A *GPU* suporta milhares de *threads* concorrentes leves e, ao contrário das *threads* da *CPU*, a sobrecarga de criação e chaveamento é insignificante. Para ocultar a alta latência da memória global, é importante ter mais *threads* do que *SPs* e ter *threads* que acessem endereços de memória consecutivos. Outro importante canal de movimentação de dados é a conexão *PCIExpress*, na qual *CPU* e *GPU* podem trocar dados entre o espaço de endereços de cada um, mas em uma velocidade muito mais lenta em comparação ao acesso a memória global. O modelo de programação da *GPU* requer que parte da

aplicação seja executada na *CPU*, enquanto a parte de computação intensiva é acelerada pela *GPU*. Um programa na *GPU* expõe o paralelismo através de uma função especial chamada *kernel*. Durante a implementação, o programador pode configurar o número de *threads* a serem usadas. As *threads* executam cálculos em paralelo e são organizadas em grupos (blocos de *threads*) que são organizados em uma estrutura de grade. Quando um *kernel* é iniciado, os blocos dentro de uma grade são distribuídos em *SMs* ociosos enquanto as *threads* são mapeados para os *SPs*. A arquitetura supracitada é comumente projetada por fabricantes como *AMD* e *NVIDIA*, esta última com o modelo de programação *CUDA*, que fornece uma extensão para *C/C++* [Franco and Bacardit 2016].

3. Classificação de Documentos

A quantidade de dados disponíveis na web continua a crescer rapidamente e a maioria destes dados corresponde a textos (documentos) que expressam a linguagem humana. Estes dados são não estruturados por natureza e para serem processados são geralmente convertidos em vetores (um por documento) que levam em conta a frequência das palavras no documento e na coleção, um valor chamado de *TF-IDF* (*Term Frequency - Inverse Document Frequency*). A representação vetorial é de alta dimensionalidade, pois depende do tamanho do vocabulário utilizado, e de alta esparsidade, pois somente uma pequena fração do vocabulário é usado em cada documento. Uma tarefa de grande importância neste contexto é a classificação, ou categorização, de documentos que normalmente é feita utilizando-se um algoritmo de aprendizado de máquina.

O aprendizado de máquina está relacionado em como programas de computador aprendem automaticamente a reconhecer padrões e tomar decisões baseadas em dados [Han et al. 2012]. Uma das subáreas do aprendizado, é a classificação, que atribui novos exemplos para categorias predefinidas. A abordagem começa com a construção de modelos a partir de conjuntos formados por muitos exemplos. Cada exemplo é caracterizado por atributos e um rótulo de classe, que é discreto para problemas de classificação e contínuo para regressão [Tan et al. 2013]. Mais formalmente, um classificador é aprendido através da análise de um conjunto de exemplos (treinamento). Um exemplo X é representado por um vetor de atributos $X = (x_1, x_2, ..., x_n)$ e está associado com um rótulo de classe predeterminado (categoria). O treinamento pode ser visto como um mapeamento ou função, $G = f(X)$, que pode predizer (classificar) o rótulo de classe G para uma amostra X. Esse mapeamento pode ser representado na forma de um modelo (regras, fórmulas ou árvores de decisão) [Han et al. 2012].

As árvores de decisão são uma das abordagens mais poderosas e populares na exploração de grandes quantidades de dados para encontrar padrões úteis [Rokach 2016]. Uma árvore é uma estrutura hierárquica consistindo de nós e arestas. Os nós internos contém condições de teste (decisões) para separar os exemplos de atributos diferentes. Os nós terminais (folhas) armazenam os rótulos de classe [Tan et al. 2013]. Vários algoritmos de indução dessas árvores foram desenvolvidos durante os anos, como o *ID3* [Quinlan 1986], o *C4.5* [Quinlan 1993] e o *CART* [Breiman et al. 1984]. Esses algoritmos particionam o conjunto de treino de forma recursiva enquanto a árvore é gerada. A ideia é selecionar o atributo que melhor discrimina os exemplos de acordo com os rótulos de classe [Han et al. 2012]. Para isso, algumas métricas avaliam a melhor divisão baseado no grau de impureza dos nós filhos. Dentre as métricas estão o ganho de informação, a entropia e o índice de *Gini*. Esta última é usada pelo *CART* [Breiman et al. 1984] e con-

sidera divisões binárias. Para dividir atributos contínuos, a condição pode ser expressada como um teste de comparação (atributo $\leq$ valor) e (atributo $>$ valor) [Tan et al. 2013].

Construir múltiplas árvores de decisão pode melhorar a capacidade de predição, ou seja, os erros cometidos por uma única árvore são compensados pelas outras [Rokach 2016]. Uma combinação de classificadores como esta é chamada de *ensemble* e serve para melhorar a acurácia da classificação. Um dos algoritmos mais populares é o Florestas Aleatórias [Breiman 2001], que cresce várias árvores até o tamanho máximo sem poda usando a metodologia *CART*. As árvores são construídas através do *bagging* [Breiman 1996] e seleção aleatória de um pequeno grupo de atributos para dividir os nós. Em resumo, ao invés de examinar todos os atributos disponíveis, a divisão é determinada pelo vetor de atributos. Depois da construção, as respostas das árvores são combinadas em um esquema de votação [Tan et al. 2013]. Outra derivação das florestas é o algoritmo de Árvores Extremamente Aleatórias [Geurts et al. 2006], que gera uma multitude de árvores de decisão a partir do conjunto de dados original usando pontos de corte aleatórios nos valores dos atributos. Da mesma forma, na classificação as predições das árvores são agregadas para produzir a predição final através da votação majoritária.

O procedimento de *bagging* (*bootstrap aggregating*) replica o conjunto de dados original muitas vezes através da escolha aleatória com substituição das amostras. Isso significa que cada amostra pode aparecer repetidas vezes no conjunto ou nenhuma. Cada árvore usa uma réplica diferente [Breiman 1996] do mesmo tamanho do conjunto original [Tan et al. 2013]. Esse procedimento faz uso de aproximadamente dois terços das amostras para treinar uma árvore. O um terço restante é referido como *out-of-bag* (*OOB*), e é usado como conjunto de teste, ou seja, a predição de uma instância acontece apenas nas árvores em que ela não foi usada para construção [James et al. 2013]. Outro tipo de *ensemble* é o *boosting*, uma estratégia que combina modelos simples (*weak*) em um processo iterativo [Freund and Schapire 1997]. Inicialmente, são atribuídos pesos iguais para todas as amostras do conjunto. Cada rodada faz uma amostragem com substituição de acordo com os pesos para formar o conjunto de treino da iteração. Assim, um classificador é derivado e avaliado a partir desses dados. Se uma amostra é classificada incorretamente, seu peso é aumentado, caso contrário, seu peso é diminuído. Além disso, os pesos de todas as amostras são normalizados para que sua soma permaneça a mesma que antes. A ideia básica é focar nas amostras mais difíceis de classificar e a complementação dos classificadores. A classificação de uma nova amostra soma os pesos de cada classificador para cada classe e a classe com a maior soma é a vencedora [Han et al. 2012]. Na maioria das situações, o tamanho das árvores em *boosting* deve ser baixo, algo entre 4 e 8 de profundidade [Hastie et al. 2009]. O *boosting* é uma das ideias mais poderosas de aprendizado, que parece dominar o *bagging* em muitos problemas, e se torna a escolha preferível [Hastie et al. 2009].

Ainda existe a combinação de *ensembles*, como o *BROOF* [Salles et al. 2015], um sistema que combina uma série de classificadores de florestas aleatórias em *boosting* para lidar com o *overfitting* causado por atributos ruidosos e irrelevantes presentes em grandes conjuntos de dados baseados na linguagem natural (texto). A estratégia usada estima o erro *out-of-bag*, produzido pelo esquema de *bagging* para atualizar os pesos das instâncias. Além disso, é feita uma atualização seletiva, em que apenas os pesos das amostras *OOB* são alterados, o que diminui o viés (*bias*) do aprendizado. A decisão fi-

nal é influenciada pela acurácia de cada floresta. Outro grupo de classificadores é o *BERT* [Campos et al. 2017], um algoritmo baseado no *BROOF*, que troca os classificadores base de florestas pelos de árvores extremamente aleatórias. A adição de aleatoriedade na escolha do ponto de corte dos valores do atributo permite diminuir a variância, fornece meios para diminuir o viés, e assim, se torna mais robusto a presença de ruídos nos dados. A ideia é produzir um modelo com alta capacidade de generalização. Além disso, a estratégia *OOB* evita que o classificador fique preso em poucos exemplos difíceis de classificar. Segundo os autores, o algoritmo está entre os melhores classificadores de texto nos conjuntos testados.

4. Trabalhos Relacionados

Algumas abordagens para acelerar árvores de decisão na *GPU* podem ser encontrados na literatura. Basicamente as implementações encontradas se diferenciam pelas diferentes estratégias de partição do problema. Em muitos trabalhos o foco está no treinamento, ou seja, na construção dos classificadores (foco deste artigo), outros já abordam apenas a tarefa da classificação em si, a inferência. A grande maioria usa o modelo de programação *CUDA*. Sharp (2018) fez uma das primeiras implementações de florestas de decisão em *GPU*. A estratégia usada envolveu mapear a estrutura de dados da floresta em um vetor de textura na memória gráfica. O treino da árvore foi feito de forma iterativa, crescendo a árvore em um nível por rodada na *GPU*. Em [Grahn et al. 2011] a abordagem utilizada foi de uma *thread* por árvore da floresta, ou seja, cada árvore foi gerada de forma sequencial. De acordo com os autores usar um número maior de árvores pode refletir em maiores ganhos.

Em [Liao et al. 2013] a construção de uma árvore de decisão acontece de forma híbrida, tudo começa por profundidade e termina por largura. A estratégia de profundidade é eficiente durante os primeiros estágios, quando grandes quantidades de exemplos são processadas. A estratégia de largura fica mais interessante quando a árvore fica profunda e é necessário avaliar menores quantidades de exemplos. Em profundidade, cada bloco é responsável por um subconjunto de exemplos de um único atributo. Depois de calcular a impureza para todos os valores, acontece uma redução para encontrar a melhor pontuação. Em largura cada bloco é responsável por um nó e todo o nível é processado simultaneamente. O foco do trabalho de [Nasridinov et al. 2013] é economizar energia acelerando a construção de uma árvore de decisão em dois níveis, nó por nó e ordenação de dados dentro do nó. [Jansson et al. 2014] usam uma estratégia em largura para gerar Árvores Extremamente Aleatórias e Florestas Aleatórias, de uma forma independente e em paralelo. Mais especificamente, a abordagem adotada funciona em nível de nó, significando que todo o nível da floresta é processado pela grade de blocos e as *threads* deslizam sobre os exemplos do nó. Ganha-se um maior potencial de paralelização de acordo com o crescimento das árvores, ou seja, quanto maior a quantidade de nós maior será a ocupação na *GPU*.

[Pianu et al. 2016] criaram uma floresta aleatória em largura por imagem para diminuir a carga de transferência entre *CPU* e *GPU*. Os autores usam *OpenCL*, por funcionar em diferentes arquiteturas. [Strnad and Nerat 2016] exploram a construção de árvores usando paralelismo em nível de nó, nível de atributo e nível de divisão. A abordagem de [Zhang et al. 2017] consiste em usar histogramas dos atributos para encontrar a melhor divisão na *GPU*, a diferença aqui é que as árvores são de regressão com *boosting*, um tipo

de *ensemble* iterativo (cada novo modelo depende do anterior). [Mitchell and Frank 2017] processam todos os nós de um nível da árvore concorrentemente na *GPU* para a estratégia *XGBoost*. Em um trabalho posterior, [Mitchell et al. 2018] usam uma abordagem *XGBoost Multi-GPU*, em que os dados de treinamento são particionados e processados por várias *GPUs*. [Wen et al. 2018] paralelizam as árvores do *XGBoost* em nível de nó, nível de atributo e nível de divisão. Por último, [Browne et al. 2018] reorganizam uma floresta de decisão pronta na memória usando um leiaute planejado para acelerar o processo de classificação. Um resumo dos trabalhos comparados pode ser conferido na Tabela 1.

Tabela 1. Comparação dos Trabalhos Correlatos.

Autores	Aprendizado	Estratégia
[Sharp 2008]	Árvore e Floresta	Por nível
[Grahn et al. 2011]	Floresta	Thread por árvore
[Liao et al. 2013]	Árvore	Bloco por exemplo e bloco por nó
[Nasridinov et al. 2013]	Árvore	Nó por nó e ordenação
[Jansson et al. 2014]	Árvore, Floresta e Árvores	Bloco por nó e janela deslizante
[Pianu et al. 2016]	Floresta	Por imagem.
[Strnad and Nerat 2016]	Árvore	Por nó, atributo e divisão
[Zhang et al. 2017]	Floresta	Histogramas dos atributos
[Mitchell and Frank 2017]	Árvore/Regressão e Boosting	Por nível
[Mitchell et al. 2018]	Árvore e XGBoost	Multi-GPU
[Wen et al. 2018]	Árvore e XGBoost	Por nó, atributo e divisão
[Browne et al. 2018]	Floresta	Reorganização

Por questões de eficiência, esta pesquisa faz uso de uma *GPU* da *NVIDIA* com *CUDA*. A estratégia usada no presente artigo é inspirada no trabalho de [Jansson et al. 2014], o único que paraleliza árvores de decisão aleatórias. Além disso, as estratégias paralelas com *boosting* da literatura usam uma única árvore de decisão. Neste trabalho é tratada da paralelização de um grupo de árvores dentro do *boosting* (*BERT*), a primeira implementação que se tem conhecimento. Tudo começa com a decomposição do problema em várias partes e mapeamento de acordo com a arquitetura da *GPU*. Em resumo, cada nível da floresta é processado por vez na grade, cada bloco é responsável por um nó da árvore, e cada *thread* cuida de um atributo ou exemplo. O projeto paralelo é discutido em maiores detalhes na próxima seção.

5. Paralelização do Algoritmo

A ideia principal é a geração das árvores aleatórias usando uma solução rápida em um ambiente mais acessível. Tudo começa com algumas configurações iniciais e alocação de recursos necessários. O primeiro passo consiste em carregar um arquivo de texto na memória principal. Como os dados de documentos textuais geralmente possuem muitos termos com valores iguais a zero, o conjunto pode ser representado de forma esparsa. Uma matriz esparsa ganha no armazenamento (apenas os valores diferentes de zero são guardados), sua desvantagem é o acesso [Wen et al. 2018]. Existem vários esquemas para armazenamento de dados esparsos com suas próprias características. O presente trabalho usa um dos formatos mais populares, o *CSR* (Linha Esparsa Compactada), que é uma forma de economizar o espaço das linhas e garantir um bom tempo de acesso. A coleção

textual consiste de exemplos que representam documentos. Cada atributo possui valores correspondentes a frequência das palavras dentro do documento.

O trabalho começa na *GPU* com a inicialização dos pesos do conjunto de dados, uma *thread* por instância. A próxima etapa faz o procedimento de *bagging*, e cada *thread* escolhe uma instância de forma aleatória de acordo com os pesos gerados no passo anterior. Assim, cada árvore recebe uma lista de instâncias novas para usar e outra com as instâncias *OOB*. A construção das árvores acontece de cima para baixo, em largura, através da divisão da lista de exemplos em cada nível até chegar nas folhas. Cada lançamento de grade (kernel) processa um nível de toda a floresta, um bloco de *threads* constrói um nó e cada *thread* cuida de um atributo. Depois, as *threads* do bloco deslizam sobre os valores dos atributos para encontrar o mínimo e o máximo para gerar o ponto de corte aleatório. Cada *thread* faz o cálculo da possível divisão baseado no atributo e no ponto gerado, como descrito no Algoritmo 1. Antes de retornar o nó, é feita uma redução para encontrar a melhor divisão. No começo de tudo é verificado se existem exemplos suficientes (*min*), se não houver, uma folha é criada. Caso contrário um nó é construído pelo bloco. No final os dados são divididos para os nós filhos. Na Figura 1 é possível ver com mais detalhes a construção de um nível intermediário da floresta.

As demais etapas, como predição, cálculo de erro, atualização e normalização usam a mesma granularidade, uma *thread* por instância. Na predição, cada *thread* classifica a instância nas árvores em que ela não foi usada (*OOB*) para construir. Depois é feita uma votação majoritária para descobrir a classe mais popular. O cálculo de erro usa a redução para somar os erros das amostras *OOB*. Na última etapa, cada *thread* atualiza o peso das amostras *OOB* mal classificadas e depois os pesos são normalizados. Um fluxograma das etapas explicadas é mostrado na Figura 2, a cor verde representa a *GPU* e a cor roxa a *CPU*. Todas as etapas são repetidas até um número definido de iterações. No final a floresta inteira é transferida de volta. Todos os dados usam vetores unidimensionais, assim, a *GPU* faz vários cálculos para acessar as posições. Alguns vetores foram pensados para levar em consideração o acesso aglutinado, já explicado anteriormente. Além disso, a movimentação entre *CPU* e *GPU* acontece apenas duas vezes, no início e no final da execução. Ao término de cada etapa, o controle volta para a *CPU* para o lançamento de nova etapa, que é processada pela *GPU*.

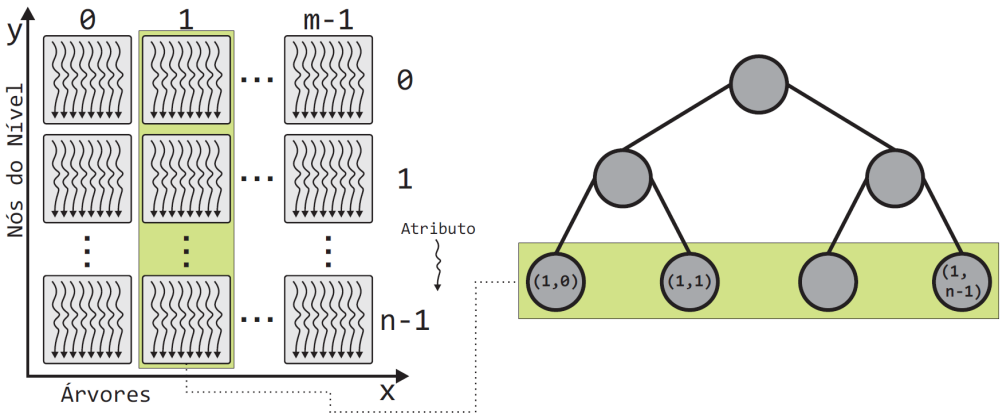

Figura 1. Construção do nível de várias árvores.

input : conjunto de treino S.
output: nó n.

if $|S| < min$ ***ou*** *a variável de saída é constante em* S **then**
  return *uma folha rotulada pelas frequências de classe*;
end
else
  for $i \leftarrow 1$ **to** K **do in parallel**
    Selecione um atributo a_i aleatoriamente;
    Encontre os valores mínimo a^S_{min} e máximo a^S_{max} de a_i;
    Faça um ponto de corte a_c aleatório entre $\left[a^S_{min}, a^S_{max}\right]$;
    Teste uma divisão s_i de a_i com a_c e calcule a pontuação;
  end
  return *nó* n*;*
end

Algorithm 1: Árvores Extremamente Aleatórias (AEA). Adaptado de [Geurts et al. 2006].

6. Experimentos e Resultados

Nesta seção são discutidos os experimentos realizados em alguns conjuntos de dados textuais usando o *BERT* sequencial e o paralelo. Os resultados obtidos exploram o uso do poder computacional das *GPUs* contrastando um processador tradicional.

6.1. Configurações de Ambiente

As configurações de hardware do computador desktop utilizado nos experimentos são descritas na Tabela 2. Todos os testes foram conduzidos em um ambiente com o sistema operacional *Linux Ubuntu 18.04 LTS* com *Kernel 4.15.0-45-generic* e *driver CUDA* versão *10* (*NVCC 10.0.130*)[1]. Cada classificador base foi composto de 8 árvores de decisão rasas, com profundidade 4. A quantidade de árvores é a mesma usada pelos autores do *BERT*. A escolha da profundidade foi explicada anteriormente. Essa configuração permite um bom uso da memória da *GPU*, uma vez que não exige grandes quantidades de armazenamento. A ideia principal é usar 200 iterações de *boosting*, mas como as iterações são realizadas repetidamente em sequência, apenas uma iteração foi levada em consideração para as estimativas do tempo de execução, já que o esforço computacional é o mesmo em cada iteração. Mesmo que a construção das árvores seja aleatória, elas estão limitadas pela profundidade.

6.2. Conjuntos de Dados

Foram usados quatro conjuntos textuais do mundo real, relacionados com artigos de ciência da computação (*ACM*), notícias (*20NG* e *Reuters*) e páginas web (*4UNI*). Todos os conjuntos foram processados previamente e adequados por [Campos et al. 2017], como remoção de *stopwords* e de alguns atributos de baixa frequência (abaixo de 6). Todos os conjuntos são compostos de vários vetores com atributos *TF-IDF*. Os conjuntos começam com baixa dimensionalidade e vão até muitas instâncias de atributos. Mais detalhes sobre os conjuntos são mostrados na Tabela 3.

[1]Baixado de *https://developer.nvidia.com/cuda-downloads* e instalado com *dpkg*.

Figura 2. Fluxo de execução do algoritmo.

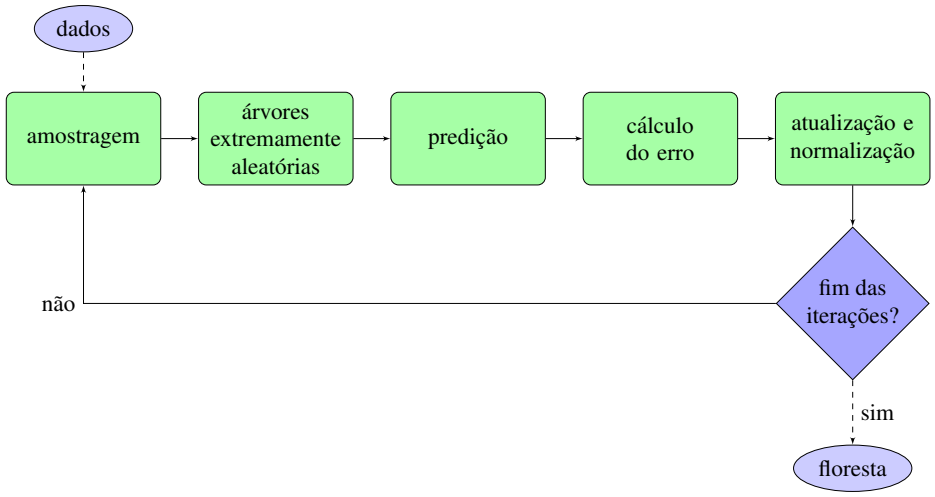

Tabela 2. Especificações de Hardware.

	CPU Intel	**GPU NVIDIA (ASUS)**
Nomenclatura	i7-7700	Expedition GeForce GTX 1070 OC edition
# de núcleos	4 e 8 *threads*	1920
Frequência do Processador	3.60 - 4.20 GHz	1582 - 1771 MHz
Arquitetura	Kaby Lake	Pascal
Memória	16 GB DDR4	8 GB GDDR5
Frequência da Memória	2400 MHz	8008 MHz

Tabela 3. Informações gerais sobre os conjuntos de dados.

Conjunto	**Classes**	**# atrib**	**# docs**	**Densidade**	**Tamanho**
4UNI	7	40,194	8,274	140.325	14 MB
ACM	11	59,990	24,897	38.805	8,5 MB
20NG	20	61,049	18,766	130.780	30 MB
REUT90	90	19,589	13,327	78.164	13 MB

6.3. Avaliação e Análise

Na avaliação da performance dos algoritmos foi comparada uma versão sequencial com a versão paralela proposta. O algoritmo sequencial foi executado em um processador tradicional. Ambas as versões foram implementadas do zero usando como base as descrições dos artigos [Campos et al. 2017, Jansson et al. 2014] e exemplos de alguns códigos. Isso é importante, uma vez que permite uma comparação mais justa. Toda a programação foi feita usando apenas a linguagem *C* e *CUDA*. Cada algoritmo foi executado 5 vezes usando conjuntos de dados diferentes. Por fim, foi feita uma média aritmética dos tempos e cálculo do desvio-padrão (DP). O resultado pode ser conferido na Tabela 4. Estima-se que executar o maior conjunto de dados com 200 iterações daria aproximadamente 9 horas de execução sequencial, uma vez que uma iteração gasta 2,65 minutos (158,99 segundos). Enquanto a execução paralela gastaria pouco mais de 20 minutos. Essa estimativa foi validada com o menor conjunto de dados. Em termos práticos, o tempo sequencial foi maior e o tempo paralelo menor que as estimativas, o que pode garantir ainda um acréscimo no fator de *speedup*.

Tabela 4. Média dos tempos de execução dos algoritmos e desvio-padrão.

Conjunto	Sequencial	DP_s	Paralelo	DP_p	*Speedup*
4UNI	15,85 s	0,17	1,27 s	0,03	12,48 x
ACM	40,84 s	0,71	2,17 s	0,01	18,80 x
20NG	113,86 s	3,05	4,90 s	0,02	23,22 x
REUT90	158,99 s	1,32	6,06 s	0,02	26,24 x

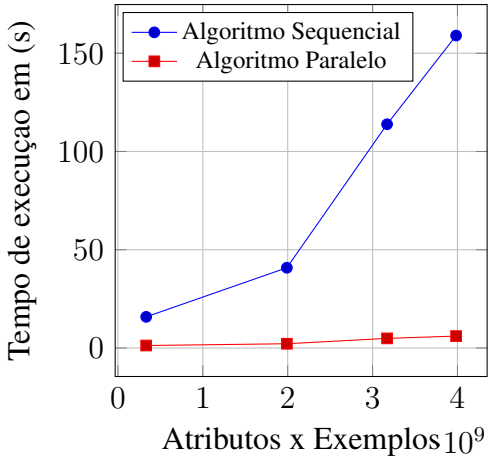

Figura 3. Execução em relação ao tamanho do conjunto.

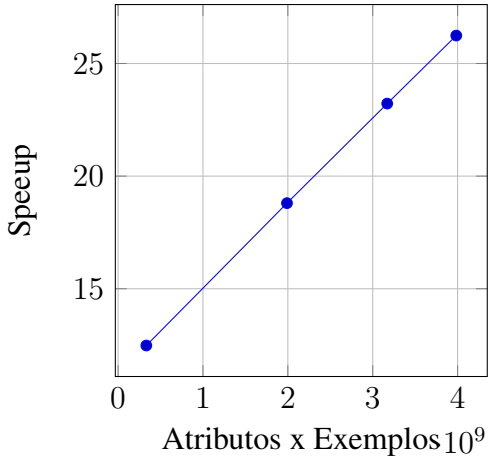

Figura 4. Speedup em relação ao tamanho do conjunto.

Como pode-se perceber no gráfico da Figura 3, quanto maior a quantidade de dados, maior é o esforço computacional sequencial, e maior ainda é o fator de *speedup* (Figura 4), ou seja, o algoritmo paralelo leva uma vantagem ainda maior quando o conjunto de dados aumenta. O maior conjunto de dados foi processado 26 vezes mais rápido com o algoritmo paralelo. Interessante ainda é o poder computacional a um custo relativamente baixo, a *GPU* usada na época da escrita deste texto custava R$ 2.200,00. Coincidentemente, uma *CPU* mais um conjunto de memórias equivalente à memória da placa gráfica custavam o mesmo valor.

7. Considerações Finais

No presente artigo foi apresentado um algoritmo paralelo para a classificação de textos. Esta é a primeira implementação paralela que usa um grupo de árvores extremamente aleatórias com a técnica de *boosting* (*BERT*) que temos conhecimento. A abordagem implementada deixa o tempo de execução até 26 vezes mais rápido que a versão sequencial nos conjuntos testados. Isso pode significar uma redução de horas para minutos. Isso é importante, uma vez que novas informações ficam disponíveis a cada dia. Os resultados mostraram também que quanto maior a dimensionalidade dos dados maior é o ganho de desempenho. Ainda assim, existem algumas melhorias para o futuro do trabalho, como testar conjuntos de dados maiores e analisar o melhor modelo paralelo com novos tipos de particionamento dos dados na *GPU*. Outras perspectivas também podem ser exploradas, como a configuração de diferentes parâmetros de execução e o uso de múltiplas placas de vídeo. O algoritmo ainda poderia ser implementado em outro ambiente de mesma instrução com vários dados (*SIMD - Single Instruction, Multiple Data*).

Agradecimentos

Ao **PIQ** do **IF Goiano** e ao grupo de pesquisa **LDA** da **UFG**. Aos revisores pelos vários questionamentos e sugestões que permitiram melhorar o artigo.

Referências

Alharthi, A., Krotov, V., and Bowman, M. (2017). Addressing barriers to big data. *Business Horizons*, 60(3):285 – 292.

Breiman, L. (1996). Bagging predictors. *Machine Learning*, 24(2):123–140.

Breiman, L. (2001). Random forests. *Mach. Learn.*, 45(1):5–32.

Breiman, L., Friedman, J. H., Olshen, R. A., and Stone, C. J. (1984). *Classification and Regression Trees*. Wadsworth and Brooks, Monterey, CA.

Browne, J., Tomita, T. M., Mhembere, D., Burns, R. C., and Vogelstein, J. T. (2018). Forest packing: Fast, parallel decision forests. *CoRR*, abs/1806.07300.

Campos, R., Canuto, S., Salles, T., de Sá, C. C., and Gonçalves, M. A. (2017). Stacking bagged and boosted forests for effective automated classification. In *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '17, pages 105–114, New York, NY, USA. ACM.

Cano, A. (2018). A survey on graphic processing unit computing for large-scale data mining. *Wiley Interdiscip. Rev. Data Min. Knowl. Discov.*, 8(1).

Chiu, C.-C., Luo, G.-H., and Yuan, S.-M. (2011). A decision tree using cuda gpus. In *Proceedings of the 13th International Conference on Information Integration and Web-based Applications and Services*, iiWAS '11, pages 399–402, New York, NY, USA. ACM.

Franco, M. A. and Bacardit, J. (2016). Large-scale experimental evaluation of gpu strategies for evolutionary machine learning. *Inf. Sci.*, 330(C):385–402.

Freund, Y. and Schapire, R. E. (1997). A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55(1):119 – 139.

Geurts, P., Ernst, D., and Wehenkel, L. (2006). Extremely randomized trees. *Machine Learning*, 63(1):3–42.

Grahn, H., Lavesson, N., Lapajne, M. H., and Slat, D. (2011). Cudarf: A cuda-based implementation of random forests. In *2011 9th IEEE/ACS International Conference on Computer Systems and Applications (AICCSA)*, pages 95–101.

Han, J., Kamber, M., and Pei, J. (2012). Data mining concepts and techniques, third edition.

Hastie, T., Tibshirani, R., and Friedman, J. (2009). *The elements of statistical learning: data mining, inference and prediction*. Springer, 2 edition.

James, G., Witten, D., Hastie, T., and Tibshirani, R. (2013). *An Introduction to Statistical Learning – with Applications in R*, volume 103 of *Springer Texts in Statistics*. Springer, New York.

Jansson, K., Sundell, H., and Boström, H. (2014). gpurf and gpuert: Efficient and scalable gpu algorithms for decision tree ensembles. In *2014 IEEE International Parallel Distributed Processing Symposium Workshops*, pages 1612–1621.

Liao, Y., Rubinsteyn, A., Power, R., and Li, J. (2013). Learning random forests on the gpu. In *Big Learning 2013: Advances in Algorithms and Data Management*, Lake Tahoe.

Mitchell, R., Adinets, A., Rao, T., and Frank, E. (2018). Xgboost: Scalable gpu accelerated learning. *CoRR*, abs/1806.11248.

Mitchell, R. and Frank, E. (2017). Accelerating the xgboost algorithm using gpu computing. *PeerJ Computer Science*, 3:e127.

Nasridinov, A., Lee, Y., and Park, Y.-H. (2013). Decision tree construction on gpu: Ubiquitous parallel computing approach. *Computing*, 96:403–413.

Navarro, C., Hitschfeld, N., and Mateu, L. (2013). A survey on parallel computing and its applications in data-parallel problems using gpu architectures. *Communications in Computational Physics*, 15:285–329.

Pianu, D., Nerino, R., Ferraris, C., and Chimienti, A. (2016). A novel approach to train random forests on gpu for computer vision applications using local features. *IJHPCA*, 30:290–304.

Quinlan, J. R. (1986). Induction of decision trees. *Mach. Learn.*, 1(1):81–106.

Quinlan, J. R. (1993). *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Rokach, L. (2016). Decision forest: Twenty years of research. *Information Fusion*, 27:111 – 125.

Salles, T., Gonçalves, M., Rodrigues, V., and Rocha, L. (2015). Broof: Exploiting out-of-bag errors, boosting and random forests for effective automated classification. In *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '15, pages 353–362, New York, NY, USA. ACM.

Sharp, T. (2008). Implementing decision trees and forests on a gpu. In Forsyth, D., Torr, P., and Zisserman, A., editors, *Computer Vision – ECCV 2008*, pages 595–608, Berlin, Heidelberg. Springer Berlin Heidelberg.

Strnad, D. and Nerat, A. (2016). Parallel construction of classification trees on a gpu. *Concurr. Comput. : Pract. Exper.*, 28(5):1417–1436.

Tan, P.-N., Steinbach, M., and Kumar, V. (2013). *Introduction to Data Mining: Pearson New International Edition (English Edition)*. Pearson Education Limited, Harlow, ESX, UK.

Wen, Z., He, B., Kotagiri, R., Lu, S., and Shi, J. (2018). Efficient gradient boosted decision tree training on gpus. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 234–243.

Zhang, H., Si, S., and Hsieh, C.-J. (2017). Gpu-acceleration for large-scale tree boosting. *CoRR*, abs/1706.08359.