Avaliação de Estilos de Código para Árvores de Decisão em GPU com Microbenchmarks

Jeronimo Penha, Alysson K. C. da Silva , Olavo Barros, Icaro Moreira, José Augusto M. Nacif, Ricardo Ferreira

[1]Universidade Federal de Viçosa (UFV) **email:** {ricardo}@ufv.br

Resumo. *Este trabalho aborda o uso de GPUs para aumentar o desempenho de algoritmos com Florestas Aleatórias (Random Forests). O estudo utiliza microbenchmarks desenvolvidos para a avaliação da implementação de árvores de decisão em GPUs, com a conclusão de que, até a profundidade de 6 níveis, a implementação sem instruções de desvio é mais vantajosa, porém para profundidades maiores, o uso de desvio, mesmo em presença de divergências, é mais indicado. O uso de implementações com memória apresenta perda de desempenho devido às indireções e latência maior que 20 ciclos de leitura em memória. Além disso, verificou-se que mais árvores com uma profundidade menor são mais eficientes do que poucas árvores com maior profundidade.*

1. Introdução

Aceleradores como GPUs (*Graphic Processing Units*) [Jansson et al. 2014, Cano 2018] e FPGAs (*Field-Programmable Gate Arrays*) [Van Chu et al. 2021] são amplamente utilizados para a melhoraria do desempenho na execução de algoritmos de aprendizado de máquina. As florestas aleatórias ou *Random Forest* (RF) são uma abordagem bastante utilizada devido à sua robustez e flexibilidade. No entanto, a implementação direta das RFs apresenta problemas de divergência de desvios e acesso desordenados aos dados.

Para que se possa obter um bom desempenho em aceleradores baseados em GPU, é usualmente recomendado evitar ao máximo a divergência de comandos condicionais como "$if - else$". Uma alternativa é a utilização de tabelas, porém o uso de indireção em memória também gera atrasos. Mesmo para a memória compartilhada com o padrão de acesso ideal, a latência é de 30 ciclos de relógio. As GPUs aproveitam várias *threads* para ocultar a latência [Volkov 2010], permitindo a exploração do escalonamento em nível de instruções de máquina. Outra alternativa é explorar a alta disponibilidade de unidades de cálculo, quando usadas sem dependências de dados. Neste trabalho apresentamos uma metodologia para geração de código para árvores de decisão.

Apesar das diretrizes da Nvidia recomendarem evitar as divergências, dependendo do problema a ser solucionado, pode ser que um código com *if* aninhados ofereça mais desempenho. Neste trabalho desenvolveu-se um conjunto de *micro-benchmarks* para avaliar a implementação de árvores de decisão em GPU. Como contribuição, são mostradas as melhores técnicas em função da profundidade da árvore. O uso dos *micro-benchmarks* como já ilustrado em outros trabalhos [Volkov 2010, Jia et al. 2019] permite a avaliação do impacto do hardware heterogêneo das GPUs para uma melhor exploração do mesmo. Os resultados deste estudo mostram que até uma profundidade de 6 níveis, o uso da implementação sem desvio, proposta neste trabalho, é mais vantajosa. Além disso, mostra-se que o uso de tabelas em memória, devido a latência, degrada mais o desempenho que os

desvios. Por fim, é constatado também que mais árvores com profundidade menor é mais eficiente que poucas árvores com uma profundidade maior.

A seção 2 apresenta os conceitos básicos. A seção 3 descreve as três implementações: if aninhados, codificação em memória e a nova codificação sem if, introduzida neste trabalho. Os experimentos com os *micro-benchmarks* são apresentados na seção 4. Finalmente, os trabalhos correlatos e as conclusões são apresentados nas seções 5 e 6.

2. Fundamentos

2.1. Florestas Aleatórias ou Random Forest

Uma árvore de decisão é um modelo de aprendizado de máquina no qual cada nó interno corresponde a um atributo do dado e cada ramo representa uma decisão com base nesse atributo. Os nós folha representam as classes. A (*Random Forest* – RF) cria várias árvores de decisão sendo que cada árvore é construída com uma amostra aleatória dos dados de treinamento. Durante a inferência, as respostas são combinadas, onde o "voto majoritário" produz a resposta final. A RF oferece robustez e generalização em relação a uma única árvore de decisão. A inferência em uma árvore é a base para a inferência na RF.

2.2. Divergência em GPU

A GPU executa no modelo SIMT (*Single Instruction, Multiple Thread*), onde um conjunto de *threads* é agrupado em uma unidade chamada *warp*. A divergência de *IF* ocorre quando diferentes *threads* dentro do mesmo *warp* seguem caminhos de execução diferentes. Algumas *threads* executam o bloco do "if" e outras o bloco do "else", que pode levar a uma diminuição significativa no desempenho. No pior caso de divergência, a GPU precisa executar cada caminho sequencialmente, uma *thread* por vez dentro de um *warp* com 32 *threads*. Idealmente, todas as *threads* do *warp* devem seguir o mesmo caminho, evitando que a GPU precise fazer comutações sequenciais entre as *threads* do *warp*.

2.3. Memórias em GPU

As GPUs possuem diferentes tipos de memória. A memória global armazena dados que precisam ser acessados por todas as *threads* em execução, possui uma latência de 200 a 800 ciclos mas sua vazão pode chegar a 1 TB/s se o padrão de acesso foi aglutinado. A memória compartilhada só existe durante a execução do kernel e apenas as *threads* dentro do bloco podem acessá-la. Sua latência é em torno de 20 a 30 ciclos. A cache *read-only* é otimizada para acessar constantes e tabelas de pesquisa, liberando a cache L1. A memória de constante é outra alternativa para tabelas. Finalmente, os registradores são a memória mais rápida, porém não permitem indexação ou indireção para armazenar tabelas. Variáveis locais com indexação serão armazenadas temporariamente na memória global com auxílio da cache para reduzir o tempo de acesso. Em termos de valores, a memória global das GPUs K80, T4 e V100 tem uma latência de 235, 220 e 215 ciclos e uma vazão de 191 GB/s, 220GB/s e 750 GB/s, respectivamente. Estes valores foram extraídos de [Jia et al. 2019]. Já para memória compartilhada, a latência é 26, 19 e 19 ciclos, respectivamente. Para a *cache read-only* a latência é 35, 92 e 89 ciclos.

2.4. Falácias em GPU e Micro-benchmarks

Volkov mostrou que, ao contrário das diretrizes do manual de melhores práticas da Nvidia [Volkov 2010] que recomendava o uso maciço de milhares de *threads*, o desempenho

pode ser melhorado com poucas *threads* e muita carga de trabalho. O objetivo é maximizar o uso de registradores e oferecer um potencial de paralelismo em nível de instrução. A análise detalhada com micro-benchmarks [Jia et al. 2019] mostra vários pontos das GPUs que devem ser observados para o desenvolvimento de códigos eficientes.

2.5. Assembly PTX

O PTX (*Parallel Thread Execution*) é uma linguagem assembly intermediária da Nvidia, usada para representar o código da GPU nas arquiteturas CUDA. O PTX implementa instruções SIMD (*Single Instruction, Multiple Data*), oferecendo instruções específicas para acesso à memória global, compartilhada e de constante, registradores e outros tipos de memória. O controle de fluxo é implementado com instruções predicativas.

3. Implementações

A Figura 1 apresenta um exemplo de uma árvore com profundidade 2 e as três implementações avaliadas: Com *if*, Sem *if* e com Tabela. A Figura 1(a) ilustra a árvore genérica de decisão com 4 folhas e três vértices de decisão. A Figura 1(b) apresenta a implementação com o pseudo código com *if's* aninhados. O mesmo código pode ser reescrito sem *if's* como ilustrado na Figura 1(c). A ideia introduzida por este trabalho utiliza a indexação nas folhas. Para simplificar a explicação, considere a árvore completa e balanceada da Figura 1(a). Neste exemplo cada folha tem seu ID, que varia de 0 à 3. Este ID irá buscar a classe indicada pela folha na implementação de uma árvore de decisão. Porém, enquanto o código com *if* percorre apenas um caminho da raiz até a folha executando uma comparação por nível, o código sem *if* executa todas as comparações da árvore. No exemplo da Figura 1(c), tem-se apenas três nós internos e portanto, três comparações. O resultado da avaliação será o número da folha. Suponha que temos apenas uma classificação binária com duas classes. Suponha que o código com if mostra, por exemplo, a classe 0 nas folhas 0 e 3 e a classe 1 nas folhas 1 e 2. A versão sem if terá o vetor de folhas $\boxed{0}\boxed{1}\boxed{1}\boxed{0}$ para este exemplo. Com a execução da operação "$vetor >> NumeroDaFolha$ and 1"teremos a classe resultante.

Finalmente, Figura 1(d) mostra a árvore codificada no formato de tabela. A linha da tabela é o ID do vértice da árvore. A tabela possui um campo para o atributo, um campo para o valor do limiar ou índice do limiar, um campo para o ramo da esquerda e outro para o ramo da direita. Caso seja uma folha, a tabela pode compartilhar o espaço e armazenar a classe no espaço dos índice dos ramos da esquerda e direita. Usando operações de deslocamento e máscara, pode-se agrupar a informação em poucas palavras de memória. Na tabela, começando pela linha 0 que é a raiz, o código para cada *thread* é o mesmo, o que evita divergências, porém a sequência de acesso as linhas poderá ser diferente para cada *thread* que irá gerar endereços diferentes de memória.

A implementação com tabela em memória, à primeira vista, parece ser a mais promissora, sendo usada em vários trabalhos com FPGA [Zhu et al. 2022] e alguns com GPU [Xie et al. 2021], pois evita as divergências da implementação com if e não necessita visitar toda a árvore como a implementação sem if, uma vez que apenas visita o caminho em profundidade da raiz até a folha. Porém, apesar da GPU oferecer várias opções de memória, todas têm uma latência de 20 ciclos ou mais. Mesmo com a execução do código com uma ocupação máxima de *threads* por multiprocessador, a GPU ainda

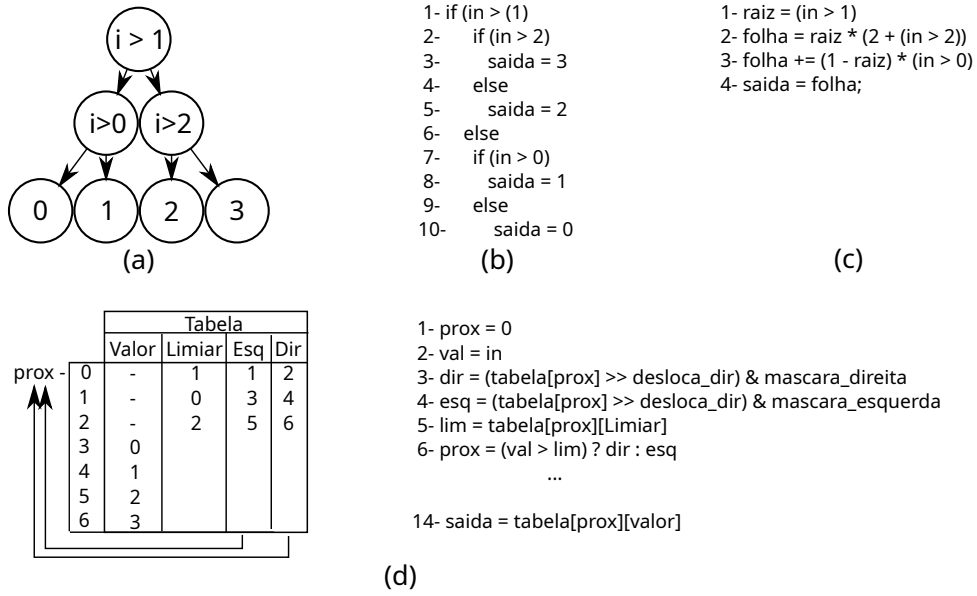

Figura 1. (a) Árvore de decisão sintética; (b) Implementação com if; (c) Sem if; (d) Tabela e código com Memória

ficará ociosa, pois não terá *threads* suficientes para esconder as latências das memórias com a especificação da árvore.

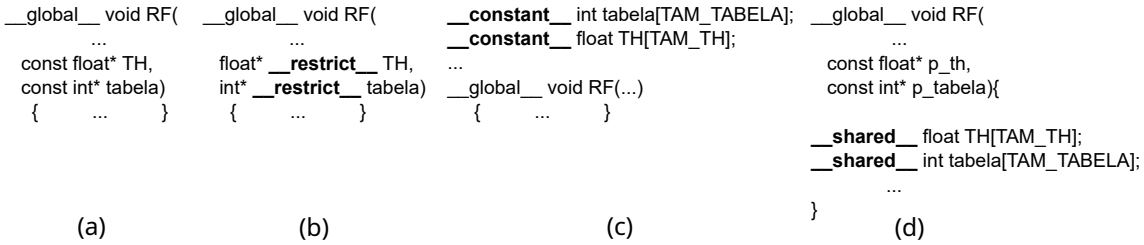

Figura 2. Declaração das memórias para implementações com tabela: (a) Global; (b) *cache read only*;(c) Constante; (d) Compartilhada

A Figura 2 mostra trechos de código para as implementações com tabela. As implementações avaliadas diferenciam pelo local onde a tabela foi armazenada: memória global, memória *read-only*, memória de constante e memória compartilhada. O código para a travessia da árvore é basicamente o mesmo, com mudanças apenas na declaração dos vetores que armazenam a tabela. Além do acesso a memória para cada vértice visitado da árvore, as implementações tem um ponto em comum: requerem mais uma indireção para acesso ao dado. A tabela pode armazenar apenas o ID do atributo e irá gerar a indireção para acessar a memória global onde estão armazenados os dados.

O código PTX é intermediário, e não necessariamente, o código binário seguirá o mesmo escalonamento. O PTX pode ser utilizado para entender as decisões do compilador [Guerreiro et al. 2019]. A Figura 3(a) mostra um pequeno trecho do código PTX gerado para a implementação com *if*. Pode-se observar que o código é bem simples, com a comparação sendo atribuída ao bit de predicado e que é utilizado pelo comando de desvio. O if com $n == 3$ é executado pela instrução *setp.eq.f32* que grava no predicado p_4.

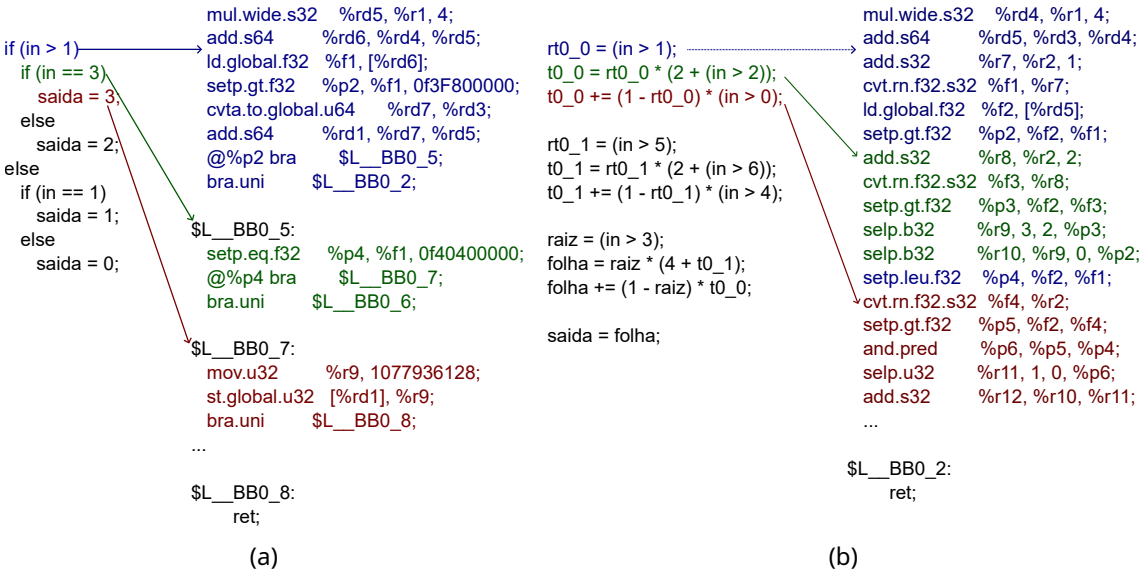

Figura 3. *Assembly* gerado para as implementações: (a) com if; (b) sem if.

Se for verdadeiro irá executar o bloco $BB0\_7$.

Já a Figura 3(b) mostra o trecho da implementação sem *if*, onde é possível observar que o bit de predicado é usado para comparação mas, as instruções de desvios são trocadas por instruções aritméticas com predicados sem a geração de divergências de controle. Não existe nenhuma instrução do tipo bra. Neste exemplo, é mostrado o código de uma árvore com 3 níveis e 8 folhas. Observe que existem 7 instruções predicativas "setp", uma para cada vértice de decisão da árvore. Este código é parametrizável e pode ter quantos níveis forem necessários.

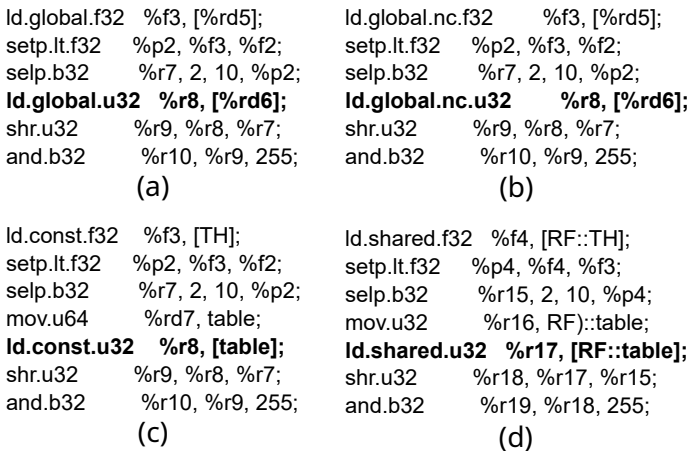

Figura 4. *Assembly* PTX para as implementações com tabela as memórias:(a) Global; (b) *cache read only*;(c) Constante; (d) Compartilhada.

Finalmente, a Figura 4 mostra o PTX das versões com tabela em memória. Pode-se observar as instruções de acesso a tabela para cada tipo de implementação avaliada. Por exemplo, o trecho da memória compartilhada usa a instrução $ld.share.f32$ e o trecho $ld.const.u32$ usa a memória de constantes.

4. Resultados

A Figura 5 mostra os tempos de execução em milisegundos para inferência nas árvores de decisão em função da profundidade. Foram avaliadas as implementações com e sem if. Para árvores bem simples, com profundidade 1 ou 2, é indiferente qual será a técnica. Árvores rasas são usadas pelos algoritmos como as abordagens de $boost$ como o Adaboost ou Gradiente Boost [Friedman 2002].

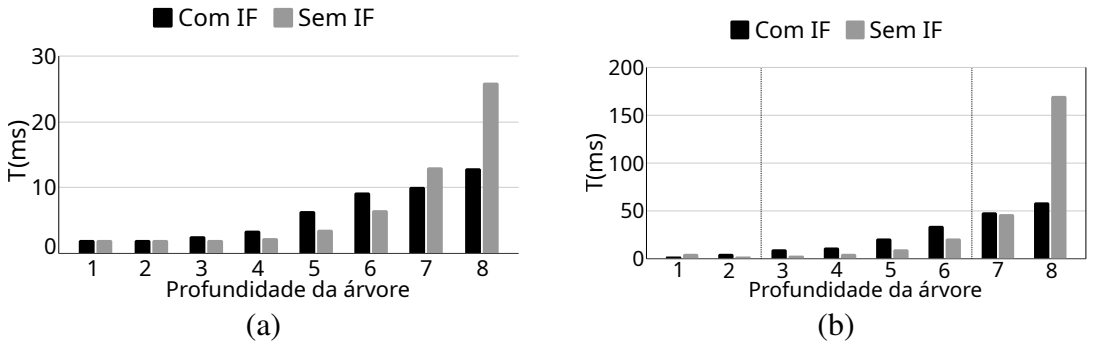

Figura 5. Tempo de execução das implementações com e sem if em função da profundidade da árvore:(a) Código com uma árvore;(b) Código com 4 árvores.

Para profundidades entre 3 a 6 níveis, a implementação sem if chega a ser até 1,8x mais rápida para uma árvore com 5 níveis, e para 4 árvores o desempenho é 3,5x melhor com 3 níveis. Com 7 níveis, a implementação sem if fica limitada pelo máximo número de operações aritmética, chegando ao limiar de ≈2 Tops/s que a GPU pode fazer. Dessa forma, mesmo com a divergência, a versão com if passa ser mais vantajosa com 7 níveis. Com 8 níveis ou mais, a versão com if é bem mais rápida devido ao crescimento exponencial do número de operações para a versão sem if. Em termos de desempenho, o tempo de execução aumenta linearmente com a quantidade de árvores. Considerando que os experimentos avaliaram 50 milhões de amostras em uma árvore com profundidade 6 que requer 6 comparações, a implementação com 4 árvores tem um desempenho aproximado de 2 G amostras/s. Se considerarmos 100 árvores para comparar com a literatura, teremos um desempenho de 80 M amostras/s.

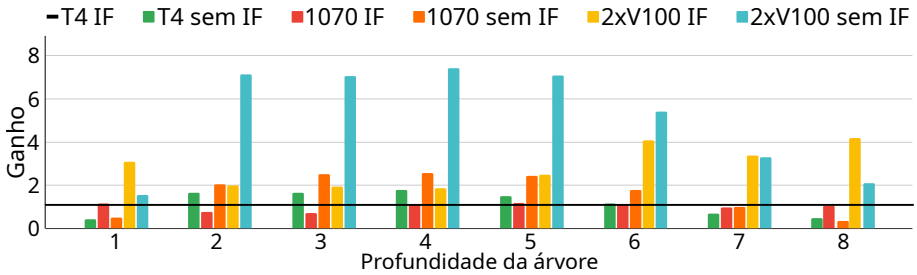

Figura 6. Avaliação do tempo de execução para diferentes GPUs com e sem if

A Figura 6 apresenta resultados do tempo de execução normalizado considerando três GPUs diferentes: T4, 1070 e V100 com 2560, 1920 e 5120 núcleos de processamento respectivamente. Os resultados estão normalizados em relação ao tempo de execução da GPU T4 para a versão com if. Para as três GPUs, o tempo da versão sem if é melhor na

faixa de 3 a 6 níveis de profundidade considerando uma floresta com 4 árvores. A GPU V100 sem if pode ser até ≈7,4x mais rápida que a GPU T4 com if.

Tabela 1. Tempo de execução (ms) para as implementações com tabela em comparação com as versões com e sem if para árvores com três níveis

Número de Árvores	Implementação com *if*	Implementação sem *if*	Implementação com Tabela							
			Constante		*Compartilhada*		*Read Only*		*Global*	
			dir	ind	dir	ind	dir	ind	dir	ind
1	2,56	1,99	2,01	2,82	2,53	3,70	6,92	8,92	8,43	8,91
2	5,21	2,13	2,40	3,78	3,20	4,96	13,90	17,67	17,06	17,66
3	7,67	2,41	2,66	6,14	3,96	6,50	25,55	26,32	25,57	22,22
4	10,12	2,89	4,67	7,75	4,65	8,08	30,95	28,99	30,93	28,54

A Tabela 1 mostra o tempo de execução das quatro implementações percorrendo uma árvore de decisão com apenas três níveis em uma tabela armazenada na memória. As GPUs oferecem várias opções de memórias. Para cada memória foram apresentados dois valores para o tempo de execução rotulados com **dir** e **ind**. As colunas **dir** mostram o tempo de execução considerando que não existe indireção para acesso aos dados e as colunas **ind** consideram a indireção. Diferente das técnicas com ou sem if, a tabela é dinâmica e não se sabe qual será o atributo de entrada em tempo de compilação. Portanto o código requer a indireção. A proposta de analisar a versão sem indireção foi implementada para separar o impacto de tabela e da indireção no tempo de execução. Pode-se observar que apenas uma indireção no acesso ao atributo é capaz de quase dobrar o tempo de execução para as implementações com memória de constante ou compartilhada. Para a versão com global com *cache* L1 ou *cache read-only* o impacto é nulo, pois já fazem acesso às caches para tabela e o acesso ao atributo fica diluído no meio dos outros acessos.

Para todas as opções avaliadas, a implementação sem if apresentou um desempenho superior. Porém, comparando com a versão com if, a implementação com a árvore armazenada na memória de constante foi até ≈2,2x mais rápida que a implementação com if para quatro árvores se fosse possível uma implementação sem indireção, mas o ganho cai para ≈1,3x quando consideramos a indireção. A implementação com memória compartilhada ($shared$) apresentou um desempenho um pouco inferior a memória de constante. A implementação com a memória *cache read-only* e a memória global (mesmo com o uso da *cache L1*) apresentaram um desempenho muito inferior.

Tabela 2. Tempo de execução (ms) para implementação com tabela em memória de constante com árvores de 7 níveis em comparação com versão com e sem if.

Número de Árvores	Com *if*	Sem *if*	*Constante*
1	10,07	13,04	13,04
2	22,07	24,18	29,46
3	35,61	34,41	43,94
4	48,74	48,34	58,43

Considerando agora 7 níveis, a Tabela 2 apresenta o tempo de execução para 1,2,3 e 4 árvores comparando o desempenho das implementações com if, sem if e com a tabela

na memória de constante. Observa-se que este cenário é o limiar no qual a implementação com if apresenta um melhor desempenho. O código foi executado na GPU 1070.

Tabela 3. Tempo de execução (ms): 5 árvores e 10 níveis; 10 árvores de 5 níveis

	Árvores	Níveis	T(ms)		Árvores	Níveis	T(ms)
Com *if*	10	5	53,31	Sem if	10	5	33,47
	5	10	127,31				

Outro ponto a ser avaliado é uma análise do tempo de execução considerando o número de árvores e suas profundidades. Como a complexidade dependente do número de comparações, n árvores com profundidade d requerem o mesmo número de comparações de d árvores com profundidade n. Neste experimento, foi avaliado que 10 árvores de 5 níveis resultam no mesmo número de comparações que a avaliação de 5 árvores de 10 níveis, totalizando no pior caso 50 comparações. A Tabela 3 mostra os resultados do tempo de execução para as implementações com if e sem if. Podemos observar que a versão sem if com 10 árvores de profundidade 5 é ≈1,6x mais rápida que a versão com if e ≈3,8x mais rápida que a versão com if usando 5 árvores de profundidade 10. Este experimento mostra que é mais interessante ter mais árvores com profundidade menor do que muitas árvores com profundidade maior. Trabalhos recentes relatam que o uso de mais árvores com a profundidade limitada é mais eficiente [Nadi and Moradi 2019], mostrando que cada árvore com a profundidade limitada é considerada como uma visão local do problema e quanto mais local a visão, melhor é a capacidade de classificação/reconhecimento. Os resultados experimentais mostraram que uma profundidade entre 5 e 7 gera uma acurácia muito próxima do máximo encontrado [Nadi and Moradi 2019]. A versão de profundidade 10 sem if não é apresentada na Tabela 3 pois tem uma grande degradação de desempenho.

A Tabela 4 mostra a divisão do trabalho de uma mesma floresta entre *threads*. Ao invés de executar as 10 árvores em uma única *thread*, duas *threads* comparam, onde uma executa 5 árvores e a outra executa 5 árvores, totalizando 10 árvores. O tempo de execução apresenta uma ligeira melhora de ≈7% em relação aos resultados apresentados na Tabela 3 que considerou a execução de todas as árvores em *thread*.

Tabela 4. Tempo de execução (ms): 5 árvores e 10 níveis; 10 árvores de 5 níveis com trabalho dividido a cada 2 *warps* com if

Árvores	Níveis	T(ms)
10	5	44,56
5	10	118,71

Uma vez validadas as implementações com dados sintéticos, o próximo passo foi a validação com dados de *datasets* da literatura. A Figura 7 mostra o tempo de execução em milissegundos da inferência nas árvores de decisão em função da profundidade com e sem instruções condicionais (*if*), utilizando os *datasets* SUSY, Adult e Hospital [Jansson et al. 2014] na GPU RTX 4090 com o emprego de 6 árvores. Nos testes realizados com os datasets Adult e Hospital, a abordagem sem o uso de instruções *if* resultou em ganhos de desempenho de até 2x com profundidade 6. Porém, pode ocorrer

casos onde os dados dos *dataset* estão organizados de forma que não tenha muita divergência, ou seja, dados "parecidos"em regiões contíguas da memória. Nestes casos, como mostra o *dataset* SUSY, com uma divergência menor, a abordagem de if foi melhor com a profundidade 6.

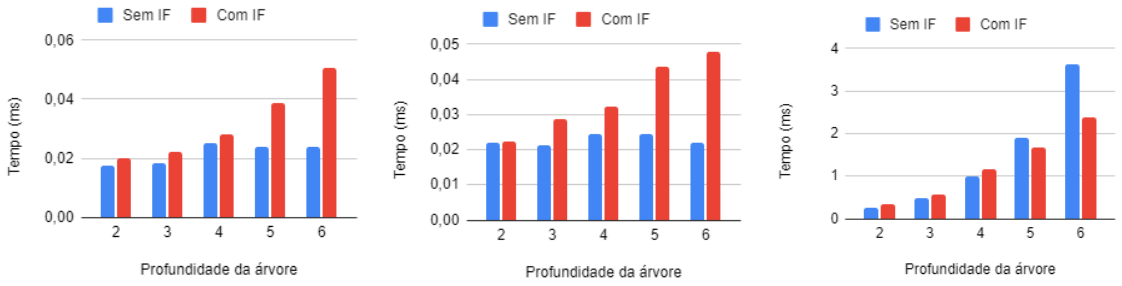

Figura 7. Tempo de execução das implementações com e sem *if* em função da profundidade da árvore:(a) *Dataset* Hospital;(b) *Dataset* Adult;(c) *Dataset* SUSY

O último experimento consiste na avaliação do comportamento e desempenho das três abordagens propostas em uma arquitetura com múltiplos núcleos. OpenMP foi usado para paralelizar as *threads*. Os processadores com múltiplos núcleos tem características bem distintas da GPU. Primeiro, cada núcleo tem uma cache L1 local com uma latência menor, em torno de 4 ciclos. Para os desvios, os processadores tem preditores dinâmicos bem avançados. Porém com relação à implementação sem desvio, como o número de unidades de cálculo é reduzido, pode ocorrer uma grande perda de desempenho, sendo esta implementação muito custosa para processadores.

O processador avaliado foi a CPU Intel(R) Core(TM) i7-7700 de 3.60GHz com 8/16 núcleos/*threads*. Selecionamos os *benchmarks* sintéticos de 5, 6 e 7 níveis com 1, 2, 3 e 4 árvores, pois foi a faixa de avaliação que permitiu comparar as três implementações das árvores de decisão. Os resultados podem ser observados na Tabela 5. Ao contrário da GPU onde a versão sem if se destacou para 5 e 6 níveis (em negrito na tabela), o código com if obteve o melhor desempenho para CPU, destacado em vermelho. Podemos observar que os preditores de desvio tem bom desempenho e que, como as execuções das *threads* são independentes, não existe o problema de divergência dos *warps*.

Nestes experimentos considerando um conjunto de 6 árvores de decisão na abordagem sem *if*, os resultados de acurácia variaram conforme o dataset analisado. Para o dataset intitulado "Hospital", a acurácia alcançada foi de 0,83. No dataset "Adult", essa métrica atingiu o valor de 0,84. Finalmente, no caso do dataset "SUSY", a acurácia foi de 0,77. Além disso, a taxa de desempenho em termos de bilhões de amostras por segundo (GAmostra/s) também foi avaliada. O dataset "SUSY"apresentou um desempenho de 13,8 GAmostra/s, enquanto que para o "Hospital"essa taxa foi de 25,1 GAmostra/s e para o "Adult"foi de 14,7 GAmostra/s.

Ao contrário de uma GPU que tem o desempenho de cálculo na ordem de Tera Flops/s, uma CPU mesmo com múltiplos núcleos, trabalha na ordem de Giga Flops/s. Portanto, a versão sem if obteve o pior desempenho devido a quantidade exponencial de cálculos em relação as versões com if ou tabela. A versão sem if na GPU (em negrito) obteve os melhores resultados para até 7 níveis dadas as características da arquitetura.

Tabela 5. Tempo de execução (ms) com/sem if e com tabela em CPU e GPU.

Árvores	Níveis	Tempo de execução (ms)					
		sem *if*		com *if*		Tabela	
		GPU	CPU	GPU	CPU	GPU	CPU
1	5	**3,62**	510,29	6,43	72,18	5,39	189,39
	6	**6,56**	1257,27	9,21	75,24	7,57	221,13
	7	13,04	2598,33	**10,07**	100,57	15,14	254,60
2	5	**5,61**	1235,78	10,88	109,70	8,23	362,16
	6	**11,55**	2620,45	15,58	148,03	13,44	423,52
	7	24,18	5102,83	**22,07**	218,01	29,67	544,08
3	5	**7,70**	1936,71	16,08	177,18	11,35	544,25
	6	**15,77**	3874,22	24,30	301,82	19,69	751,58
	7	**34,41**	7601,23	35,61	326,44	44,17	865,45
4	5	**10,23**	2572,75	21,32	310,98	14,32	799,97
	6	**21,62**	5085,65	34,42	343,99	25,87	958,26
	7	**46,95**	10111,26	48,34	440,42	58,66	1078.87

A implementação com tabelas obteve resultados de $\approx$ 2x, em média, mais lentos que a implementação com if. Mesmo a cache do processador tendo uma latência menor, como a quantidade concorrente de *threads* na GPU é maior, o código com tabela degrada mais em CPU. Entretanto, a tabela tem desempenho superior em relação a versão sem if no processador, sendo a implementação com tabela aproximadamente $\approx$ 10x mais rápida que a implementação sem if.

Por fim, podemos observar que a versão sem if da GPU, proposta neste trabalho, considerando 5 a 6 níveis, chega a ser 2x mais rápida que a a versão com if na GPU, $\approx$1,6 mais rápida que a versão com tabela em GPU. Como relação ao processador com múltiplos núcleos, a versão sem if da GPU é $\approx$15, $\approx$43 e $\approx$217x mais rápida que as versões com if, com tabela e sem if no processador.

5. Trabalhos Relacionados

Aceleradores de RF com GPU e FPGA já vem sendo propostos há duas décadas. Alguns trabalhos usam aceleradores para construção das árvores [Jansson et al. 2014, Lin et al. 2019] e outros focam na inferência para classificação ou regressão [Van Chu et al. 2021, Guan et al. 2023]. Neste artigo avaliamos o processo de inferência. Estudos recentes mostram que em ciência de dados, o tempo da inferência em relação ao treinamento vem se tornando um componente dominante (na faixa de 45% a 65%) [Nakandala 2020], motivando um estudo detalhado da inferência em árvores de decisão, que envolvem uma travessia simultânea de várias árvores de decisão [Prasad et al. 2022]. Otimizar a travessia de árvores em CPU/GPU é desafiador pois elas têm padrões de acesso irregulares, possuem uma localidade espacial e temporal fraca e, portanto, um desempenho de cache deficiente, além de divergências e dependências verdadeiras entre as instruções que causam várias paradas no pipeline [Prasad et al. 2022]. Além disso, não é trivial acelerar as travessias de árvores com otimizações de baixo nível, como a vetorização usando instruções SIMD [Jo et al. 2013]. Algumas estratégias são a reordenação dos nodos das árvores e agrupamentos para melhorar a localidade das caches e simplificar a vetorização. Além disso, intercalar a avaliação de múltiplas árvores pode reduzir as latências de dependências de dados em

modelos com tabelas em CPU [Prasad et al. 2022].

Trabalhos iniciais [Van Essen et al. 2012] mostram que otimizar o problema para GPU não é trivial e pode apresentar ganhos baixos, sendo as versões em FPGA mais flexíveis em função das divergências de controle. Os resultados iniciais [Van Essen et al. 2012] mostram um desempenho de 20 mil amostras por segundo em uma GPU Tesla 2050 com 448 núcleos. A implementação distribuiu as árvores entre as *threads* para depois reduzir e realizar a votação. Trabalhos recentes [Van Chu et al. 2021] mostram que as GPUs [Nakandala 2020] podem ter um desempenho de até 100 mil amostras por segundo com profundidade variando de 5 a 7 níveis como também sugerido em [Nadi and Moradi 2019], se considerarmos 4 árvores. Porém, o uso de FPGA com partição do espaço de atributos pode ser até uma ordem de grandeza mais rápido [Van Chu et al. 2021] em algumas situações. Assim como a partição do espaço de atributos [Van Chu et al. 2021], outras variações como o trabalho apresentado por [Zhang 2022] propõem a travessia de árvores em uma representação matricial de sua estrutura hierárquica buscando o produto interno máximo. Estas variações ainda não foram exploradas em GPU.

6. Conclusão

Este trabalho investigou o uso de GPUs para aprimorar o desempenho de algoritmos com árvores de decisão e Florestas Aleatórias. Uma contribuição foi a introdução de uma implementação estática gerada em tempo de compilação que elimina a necessidade de desvios. Entretanto, o número de operações que são executadas cresce de forma exponencial. Mesmo assim, a implementação sem desvio proposta foi vantajosa até uma profundidade de 6 níveis, que é recomendada em muitas situações [Nadi and Moradi 2019]. Para profundidades maiores, o uso de desvios mostrou-se mais indicado. A distribuição eficiente da carga de trabalho entre mais árvores com profundidade menor demonstrou ser mais eficiente do que poucas árvores com profundidade maior. Estes resultados foram obtidos através da construção e análise de micro-benchmarks para avaliar o tempo de execução em função da implementação em GPU. Eles fornecem diretrizes para otimizar a eficiência da inferência com conjuntos de árvores de decisão em aceleradores. Finalmente, apesar da GPU oferecer várias opções de memória, a modelagem de árvores em memória não apresenta desempenho, sendo mais adequada para FPGA [Wang and Jin 2022] e CPU [Prasad et al. 2022]. Trabalhos futuros podem avaliar a implementação com intercalação das árvores para esconder as latências como apresentado em [Xie et al. 2021], mas depende de um controle do escalonamento das instruções pelo compilador.

7. Agradecimentos

FAPEMIG (APQ-01577-22), CNPq, Funarbe. O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior Brasil (CAPES) Código de Financiamento 001.

Referências

[Cano 2018] Cano, A. (2018). A survey on graphic processing unit computing for large-scale data mining. *Wiley Interdisciplinary: Data Mining and Knowledge Discovery*.

[Friedman 2002] Friedman, J. H. (2002). Stochastic gradient boosting. *Computational statistics & data analysis*, 38(4):367–378.

[Guan et al. 2023] Guan, H., Min, H., Yu, L., and Zou, J. (2023). A comparison of decision forest inference platforms from a database perspective. *arXiv:2302.04430*.

[Guerreiro et al. 2019] Guerreiro, J., Ilic, A., Roma, N., and Tomas, P. (2019). Gpu static modeling using ptx and deep structured learning. *IEEE Access*.

[Jansson et al. 2014] Jansson, K., Sundell, H., and Boström, H. (2014). gpurf and gpuert: efficient and scalable gpu algorithms for decision tree ensembles. In *IPDPS*.

[Jia et al. 2019] Jia, Z., Maggioni, M., Smith, J., and Scarpazza, D. P. (2019). Dissecting the nvidia turing t4 gpu via microbenchmarking. *arXiv preprint arXiv:1903.07486*.

[Jo et al. 2013] Jo, Y., Goldfarb, M., and Kulkarni, M. (2013). Automatic vectorization of tree traversals. In *PACT*. IEEE.

[Lin et al. 2019] Lin, Z., Sinha, S., and Zhang, W. (2019). Towards efficient and scalable acceleration of online decision tree learning on fpga. In *IEEE FCCM*.

[Nadi and Moradi 2019] Nadi, A. and Moradi, H. (2019). Increasing the views and reducing the depth in random forest. *Expert Systems with Applications*, 138:112801.

[Nakandala 2020] Nakandala, S. (2020). A tensor compiler for unified machine learning prediction serving. In *Symp on Operating Systems Design and Implementation (OSDI)*.

[Prasad et al. 2022] Prasad, A., Govindarajan, R., and Bondhugula, U. (2022). Treebeard: An optimizing compiler for decision tree based ml inference. In *IEEE MICRO*.

[Van Chu et al. 2021] Van Chu, T., Kitajima, R., Kawamura, K., Yu, J., and Motomura, M. (2021). A high-performance and flexible fpga inference accelerator for decision forests based on prior feature space partitioning. In *IEEE ICFPT*.

[Van Essen et al. 2012] Van Essen, B., Macaraeg, C., Gokhale, M., and Prenger, R. (2012). Accelerating a random forest classifier: Multi-core, gp-gpu, or fpga? In *IEEE FCCM*.

[Volkov 2010] Volkov, V. (2010). Better performance at lower occupancy. In *Proceedings of the GPU technology conference, GTC*, volume 10, page 16. San Jose, CA.

[Wang and Jin 2022] Wang, H. and Jin, H. (2022). Hardgbm: A framework for accurate and hardware-efficient gradient boosting machines. *IEEE Transaction on CAD*.

[Xie et al. 2021] Xie, Z., Dong, W., Liu, J., Liu, H., and Li, D. (2021). Tahoe: tree structure-aware high performance inference engine for decision tree ensemble on gpu. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 426–440.

[Zhang 2022] Zhang, J. (2022). Rethink decision tree traversal. *arXiv preprint arXiv:2209.04825*.

[Zhu et al. 2022] Zhu, M., Luo, J., Mao, W., and Wang, Z. (2022). An efficient fpga-based accelerator for deep forest. In *ISCAS*. IEEE.