**UNIVERSIDADE FEDERAL DE VIÇOSA**

# Otimização de Modelos de Aprendizado de Máquina em Dispositivos de Borda: Random Forest e XGBoost

Olavo Alves Barros Silva
*Magister Scientiae*

**VIÇOSA - MINAS GERAIS**
**2026**

**OLAVO ALVES BARROS SILVA**

# Otimização de Modelos de Aprendizado de Máquina em Dispositivos de Borda: Random Forest e XGBoost

Dissertação apresentada à Universidade Federal de Viçosa, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, para obtenção do título de *Magister Scientiae*.

Orientador: Ricardo dos S. Ferreira

Coorientador: Jose Augusto Miranda Nacif

**VIÇOSA - MINAS GERAIS**
**2026**

**Ficha catalográfica elaborada pela Biblioteca Central da Universidade Federal de Viçosa - Campus Viçosa**

T
S586o
2026

Silva, Olavo Alves Barros, 1998-
Otimização de modelos de aprendizado de máquina em dispositivos de borda: Random Forest e XGBoost / Olavo Alves Barros Silva. – Viçosa, MG, 2026.
1 dissertação eletrônica (87 f.): il. (algumas color.).

Orientador: Ricardo dos Santos Ferreira.
Dissertação (mestrado) - Universidade Federal de Viçosa, Departamento de Informática, 2026.
Referências bibliográficas: f. 83-87.
DOI: https://doi.org/10.47328/ufvbbt.2026.377
Modo de acesso: World Wide Web.

1. Aprendizado do computador. 2. Processamento em dispositivos de borda (Computação). I. Ferreira, Ricardo dos Santos, 1969-. II. Universidade Federal de Viçosa. Departamento de Informática. Programa de Pós-Graduação em Ciência da Computação. III. Título.

CDD 22. ed. 006.31

Bibliotecário(a) responsável: Euzébio Luiz Pinto CRB-6/3317

**OLAVO ALVES BARROS SILVA**

**Otimização de Modelos de Aprendizado de Máquina em Dispositivos de Borda: Random Forest e XGBoost**

Dissertação apresentada à Universidade Federal de Viçosa, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, para obtenção do título de *Magister Scientiae*.

APROVADA: 24 de fevereiro de 2026.

Assentimento:

_____

Olavo Alves Barros Silva
Autor

_____

Ricardo dos Santos Ferreira
Orientador

Dedico este trabalho aos meus pais, Gilberto e Maria do Carmo, grandes incentivadores que sempre me fizeram acreditar na realização deste sonho e batalharam junto a mim para realizá-lo.

## AGRADECIMENTOS

Agradeço a Deus por todos os objetivos conquistados com muito esforço e perseverança, e por manter-me sempre confiante de que o estudo é a melhor forma de ascender social, intelectual e economicamente.

Aos meus pais, Gilberto e Maria do Carmo, por estarem, incondicionalmente, ao meu lado em todos os momentos.

À Universidade Federal de Viçosa - UFV e a todos os professores do Programa de Pós-Graduação em Ciência da Computação que contribuíram imensamente nos meus estudos.

Ao professor Ricardo dos Santos Ferreira pela prestimosa colaboração e paciência em me orientar.

Aos meus amigos e colegas de laboratório, os quais contribuíram muito para a conclusão do mestrado, pois tenho certeza de que não conseguiria sozinho.

A todos os pesquisadores e colaboradores que auxiliaram direta ou indiretamente no desenvolvimento deste trabalho, pela inestimável parceria.

Este trabalho foi realizado com o apoio das seguintes agências de pesquisa brasileiras: Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Código de Financiamento 001, Fundação de Amparo à Pesquisa do Estado de Minas Gerais (FAPEMIG) e Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq).

Projeto FAPEMIG - EDITAL 001/2022 - DEMANDA UNIVERSAL, APQ-01577-22 PROJETO: “MAPEAMENTO DE ACELERADORES DE ALTO DESEMPENHO EM ARQUITETURAS DE DOMÍNIO ESPECÍFICO”.

"Ah, o primeiro é sempre o mais difícil, criança! Agora você sabe como se faz! Daqui para a frente, só fica mais fácil. [...] E quando se cansar, tire um momento para olhar para essa peça novamente. Lembre-se sempre que precisar: quem foi que a fez, e a fez tão bem?"

— Kaoru Mori, Otoyomegatari (A Bride's Story)

## RESUMO

SILVA, Olavo Alves Barros, M.Sc., Universidade Federal de Viçosa, fevereiro de 2026. **Otimização de Modelos de Aprendizado de Máquina em Dispositivos de Borda: Random Forest e XGBoost**. Orientador: Ricardo dos Santos Ferreira. Coorientador: Jose Augusto Miranda Nacif.

A crescente demanda por processamento inteligente em dispositivos de borda (edge computing) tem impulsionado o desenvolvimento de técnicas de Tiny Machine Learning (TinyML), que visam executar algoritmos de aprendizado de máquina em plataformas com severas restrições de energia, memória e capacidade computacional. Neste contexto, os Field-Programmable Gate Arrays (FPGAs) emergem como uma alternativa promissora, oferecendo equilíbrio único entre eficiência energética, desempenho e reconfigurabilidade. No entanto, a implementação eficiente de modelos de aprendizado de máquina em FPGAs de borda enfrenta desafios significativos, incluindo a lacuna entre ferramentas de alto nível como scikit-learn e XGBoost e as descrições de hardware de baixo nível necessárias para programação de FPGAs (como Verilog e VHDL). Esta dissertação propõe um conjunto integrado de quatro ferramentas que automatiza o fluxo completo desde o modelo treinado até o circuito otimizado em FPGA, aplicando otimizações em múltiplos níveis: poda de árvores, quantização durante o mapeamento e poda de circuitos a nível de Look-up Tables (LUTs).

A primeira ferramenta, XGB2GPU, realiza exploração paralela do espaço de poda de modelos XGBoost utilizando GPUs, implementando três estratégias topológicas (Flat, Linear e Exponencial) e alcançando ganho de desempenho de 900× a 3000× em comparação com implementações em CPU.

A segunda ferramenta, RDSF, mapeia modelos Random Forest em FPGAs utilizando Binary Decision Diagrams (BDDs) para otimização de funções Booleanas.

A terceira ferramenta, TreeLUT, estendida neste trabalho, mapeia modelos XGBoost aplicando quantização em duas etapas (dados de entrada e pesos das folhas), incluindo contabilização completa dos recursos de hardware (módulos de quantização e argmax) com geração eficiente do código Verilog. A implementação das árvores baseada em multiplexadores reduziu o uso de LUTs em aproximadamente 20% comparada à abordagem por equações.

A quarta ferramenta, Go-Fast, implementa simulação aproximada acelerada por GPU para poda de circuitos baseados em LUTs, alcançando ganhos de desempenho de cinco ordens de magnitude (10^5) em comparação com simuladores de Verilog do estado da arte como Verilator.

Os resultados demonstraram que é possível alcançar reduções de área: até 90% via poda de árvores mantendo degradação de acurácia inferior a 5% em alguns casos, e até 41% via poda de circuitos com erro controlado. Adicionalmente, demonstrou-se que simulação com grandes volumes de estímulos (2^30 amostras) revela configurações superiores às encontradas com conjuntos reduzidos (2^14 amostras), evidenciando a importância da amostragem ampla.

A principal contribuição deste trabalho reside na demonstração de que otimizações em diferentes níveis de abstração (modelo, mapeamento e circuito) podem ser compostas de forma conjunta para alcançar implementações eficientes, reduzindo significativamente a barreira entre ferramentas de alto nível e implementações em hardware reconfigurável para aplicações de TinyML.

Palavras-chave: Tiny Machine Learning; FPGA; XGBoost ; Random Forest ; Poda de Árvores ; Computação Aproximada ; GPU ; Otimização de Hardware.

## ABSTRACT

SILVA, Olavo Alves Barros, M.Sc., Universidade Federal de Viçosa, February, 2026. **Optimizing Machine Learning Models on Edge Devices: Random Forest e XGBoost**. Adviser: Ricardo dos Santos Ferreira. Co-adviser: Jose Augusto Miranda Nacif.

The growing demand for intelligent processing in edge computing devices has driven the development of Tiny Machine Learning (TinyML) techniques, which aim to execute machine learning algorithms on platforms with severe energy, memory, and computational capacity constraints. In this context, Field-Programmable Gate Arrays (FPGAs) emerge as a promising alternative, offering a unique balance between energy efficiency, performance, and reconfigurability. However, the efficient implementation of machine learning models on FPGAs faces significant challenges, including the gap between high-level tools like scikit-learn and XGBoost and the low-level hardware descriptions required for FPGA programming (such as Verilog and VHDL). This dissertation proposes an integrated set of four tools that automates the complete flow from the trained model to the optimized circuit on FPGA, applying optimizations at multiple levels: tree pruning, quantization during mapping, and circuit pruning at the Look-up Tables (LUTs) level.

The first tool, XGB2GPU, performs parallel exploration of the XGBoost model pruning space using GPUs, implementing three topological strategies (Flat, Linear, and Exponential) and achieving speedups of 900x to 3000x compared to CPU implementations.

The second tool, RDSF, maps Random Forest models onto FPGAs using Binary Decision Diagrams (BDDs) for Boolean function optimization.

The third tool, TreeLUT, extended in this work, maps XGBoost models by applying two-stage quantization (input data and leaf weights), including full accounting of hardware resources (quantization modules and argmax).

The fourth tool, Go-Fast, implements GPU-accelerated approximate simulation for LUT-based circuit pruning, achieving speedups of five orders of magnitude (10^5) compared to traditional simulators like Verilator.

The results demonstrated that substantial area reductions are achievable: up to 90% via tree pruning while maintaining accuracy degradation below 5% in some cases, and up to 41% via circuit pruning with controlled error.

The multiplexer-based implementation reduced LUT usage by approximately 20% compared to the equation-based approach. Additionally, it was demonstrated that simulation with large volumes of stimuli (2^30 samples) reveals configurations superior to those found with reduced sets (2^14 samples), highlighting the importance of broad sampling.

The main contribution of this work lies in demonstrating that optimizations at different abstraction levels (model, mapping, and circuit) can be jointly composed to achieve efficient implementations, significantly reducing the barrier between high-level tools and reconfigurable hardware implementations for TinyML applications.

Keywords: Tiny Machine Learning; FPGA ; XGBoost; Random Forest; Tree Pruning; Approximate Computing; GPU; Hardware Optimization

# Lista de ilustrações

# Lista de tabelas

# Sumário

# 1 Introdução

## 1.1 Contextualização

Nas últimas décadas, a evolução tecnológica impulsionou uma transformação digital sem precedentes, marcada pela proliferação de dispositivos inteligentes e pela conectividade global. Este fenômeno, consolidado no conceito de Internet das Coisas (*Internet of Things* - IoT), resultou em bilhões de objetos físicos conectados à rede mundial, gerando um volume de dados (*Big Data*) que necessitam de processamento e análise contínua (ABADADE et al., 2023). Estima-se que o número de dispositivos IoT conectados ultrapasse 100 bilhões (KAVRE; GADEKAR; GADHADE, 2019), criando desafios sem precedentes em termos de processamento distribuído e eficiência energética.

Tradicionalmente, a arquitetura de processamento desses dados baseava-se quase exclusivamente na Computação em Nuvem (*Cloud Computing*). Neste modelo, os dados coletados pelos sensores são transmitidos para servidores centralizados para processamento e armazenamento, tendo, portanto, um foco maior em latência e vazão de dados (LIN et al., 2023). Contudo, com o crescimento exponencial do número de dispositivos e a demanda por aplicações de tempo real, esse modelo centralizado começou a apresentar gargalos significativos, como latência elevada, alto consumo de largura de banda, preocupações com privacidade e segurança dos dados, além de dependência crítica de conectividade estável (DUTTA; BHARALI, 2021).

Como resposta a essas limitações, surgiu o paradigma da Computação de Borda (*Edge Computing*), que propõe aproximar o processamento da fonte geradora dos dados. Dentro deste espectro, destaca-se o *Tiny Machine Learning* (TinyML), uma área de pesquisa focada em habilitar a execução de algoritmos de aprendizado de máquina em microcontroladores e dispositivos de ultra-baixo consumo energético (na ordem de miliwatts). O TinyML viabiliza a inteligência local, permitindo que dispositivos tomem decisões autônomas sem a necessidade constante de comunicação com a nuvem, o que é crucial para aplicações críticas, remotas e sensíveis à latência (CAPOGROSSO et al., 2024).

No contexto da implementação de TinyML, três plataformas principais emergem como candidatas para a execução eficiente de modelos de aprendizado de máquina: microcontroladores (MCUs), Circuitos Integrados de Aplicação Específica (*Application-Specific Integrated Circuits* - ASICs) e *Field-Programmable Gate Arrays* (FPGAs). Cada uma dessas plataformas apresenta características distintas em termos de flexibilidade, eficiência energética e complexidade de desenvolvimento.

Os MCUs (SANCHEZ-IBORRA; SKARMETA, 2020; IMMONEN; HÄMÄLÄINEN, 2022; GIORDANO; PICCINELLI; MAGNO, 2022) oferecem facilidade de programação através de linguagens de alto nível e ampla disponibilidade de ferramentas de desenvolvimento, porém apresentam limitações significativas em termos de desempenho para modelos complexos. Os ASICs, por outro lado, proporcionam o máximo de eficiência energética e desempenho(KALAPOTHAS et al., 2023), mas exigem investimentos substanciais e carecem de flexibilidade pós-fabricação.

Neste cenário, os FPGAs emergem como uma alternativa promissora, oferecendo um equilíbrio único entre eficiência energética, desempenho e reconfigurabilidade (BOUTROS; ARORA; BETZ, 2025). A capacidade de reconfiguração dos FPGAs permite a implementação de arquiteturas otimizadas especificamente para os modelos de ML desejados, proporcionando eficiência comparável aos ASICs mantendo a flexibilidade de atualização e adaptação. Adicionalmente, a natureza paralela intrínseca dos FPGAs alinha-se perfeitamente com as operações massivamente paralelas características dos algoritmos de aprendizado de máquina, especialmente em modelos baseados em árvores de decisão como *Random Forest* e *XGBoost* (ALCOLEA; RESANO, 2021; GAJJAR et al., 2022; SUMMERS et al., 2020; KHATAEI; BAZARGAN, 2025).

## 1.2 Justificativa

No contexto atual, no qual a Internet das Coisas se tornou parte integrante do cotidiano, com dispositivos conectados de forma contínua e novos paradigmas emergindo em áreas como saúde, agricultura de precisão, cidades inteligentes e transporte (ABADADE et al., 2023), observa-se uma crescente dependência de técnicas de *Machine Learning* para permitir que esses dispositivos tomem decisões localmente e em tempo real.

Nesse sentido, o TinyML, desde sua consolidação em 2019 (CAPOGROSSO et al., 2024), tem experimentado crescimento substancial, permitindo a criação de modelos de ML com baixo consumo energético, otimizados para a computação de borda. No entanto, uma característica intrínseca aos dispositivos de borda é a limitação em termos de capacidade computacional. Além disso, a maioria dos estudos existentes concentra-se em redes neurais profundas, deixando em segundo plano modelos clássicos baseados em árvores de decisão, que apresentam vantagens significativas para implementação em hardware reconfigurável (DONG et al., 2020).

Apesar do potencial dos FPGAs para TinyML, existe uma lacuna significativa entre as ferramentas de desenvolvimento de alto nível utilizadas por cientistas de dados (como *scikit-learn* e *XGBoost*) e as descrições de hardware de baixo nível necessárias para programação de FPGAs (como Verilog e VHDL). Esta lacuna representa uma barreira substancial para a adoção ampla de FPGAs em aplicações de TinyML, exigindo

conhecimento especializado em projeto de sistemas digitais que está além do escopo de especialistas em aprendizado de máquina.

Esta pesquisa se justifica tanto do ponto de vista científico quanto prático. Do ponto de vista científico, busca-se desenvolver técnicas automatizadas para conversão de modelos de árvores de decisão em implementações eficientes em FPGA, reduzindo a barreira de entrada para cientistas de dados. Além disso, exploram-se estratégias de otimização em múltiplos níveis: desde o modelo treinado por meio de poda de árvores, passando pela quantização durante o mapeamento, até a otimização do circuito final via poda de LUTs. Investiga-se também o uso de GPUs não apenas para treinamento, mas como plataforma de exploração massiva do espaço de projeto para otimizações de poda e simulação aproximada. Por fim, contribui-se para o corpo de conhecimento sobre trade-offs entre acurácia e área de hardware em implementações de TinyML baseadas em FPGA.

Do ponto de vista prático, os resultados podem ser aplicados diretamente em sistemas IoT de grande impacto, como dispositivos de monitoramento remoto em redes de sensores com restrições energéticas severas, sensores agrícolas operando em locais com conectividade limitada ou inexistente, sistemas de diagnóstico médico portátil para regiões com infraestrutura limitada, e dispositivos industriais de manutenção preditiva com requisitos de baixa latência e alta confiabilidade.

Além disso, no contexto de aplicações de aprendizado de máquina, a seleção de atributos relevantes e a redução da dimensionalidade dos dados permitem que modelos com complexidade reduzida se equiparem a modelos mais complexos (TABANELLI; TAGLIAVINI; BENINI, 2023). A utilização de técnicas de quantização como etapa de pré-processamento pode ser uma estratégia eficaz para atingir esse objetivo, reduzindo a demanda por processamento na etapa de inferência.

Por fim, a natureza dos modelos baseados em árvores de decisão apresenta uma característica única e vantajosa para implementação em hardware: sua estrutura pode ser naturalmente mapeada em funções Booleanas e, consequentemente, em *Look-up Tables* (LUTs), os elementos lógicos reconfiguráveis dos FPGAs. Esta propriedade permite uma implementação eficiente, evitando a complexidade associada à implementação de operações aritméticas de ponto flutuante necessárias em redes neurais profundas.

## 1.3 Objetivos da Pesquisa, Perguntas e Hipóteses

### 1.3.1 Pergunta de Pesquisa

Como desenvolver uma metodologia de fluxo automatizado, através da implementação de ferramentas interconectadas, que permita a construção eficiente de modelos baseados em árvores de decisão (Random Forest e XGBoost) em FPGAs para aplicações de

TinyML, mantendo a acurácia da inferência enquanto supera as limitações de capacidade computacional, memória e consumo energético inerentes aos dispositivos de borda?

### 1.3.2 Hipótese

Com base na pergunta de pesquisa e na análise do estado da arte, formula-se a seguinte hipótese central:

É possível desenvolver um conjunto integrado de ferramentas que automatize o mapeamento de modelos baseados em árvores de decisão (Random Forest e XGBoost) em FPGAs, aplicando otimizações em múltiplos níveis do fluxo de desenvolvimento (poda de árvores, quantização e poda de circuito), de modo que:

**H1:** A exploração massivamente paralela do espaço de poda utilizando GPUs permite identificar configurações que reduzem significativamente o número de nós nas árvores sem degradação da acurácia;

**H2:** A aplicação de técnicas de quantização aos dados de entrada e aos pesos das folhas permite reduzir os recursos usados nos FPGA (medida em número de LUTs) sem comprometer a acurácia;

**H3:** A simulação aproximada acelerada por GPU viabiliza a exploração de estratégias de poda a nível de circuito (LUTs), permitindo reduções adicionais dos recursos de hardware para aplicações tolerantes a pequenos erros de aproximação;

**H4:** O conjunto de ferramentas desenvolvido demonstra ganhos de desempenho em comparação com abordagens baseadas em CPU para as tarefas de exploração do espaço de projeto e simulação de circuitos.

### 1.3.3 Objetivo Geral

Desenvolver e validar um conjunto integrado de ferramentas automatizadas que viabilize a implementação eficiente de modelos baseados em árvores de decisão (Random Forest e XGBoost) em FPGAs para aplicações de TinyML, incorporando técnicas de otimização em múltiplos níveis do fluxo de desenvolvimento.

### 1.3.4 Objetivos Específicos

Para alcançar o objetivo geral, foram definidos os seguintes objetivos específicos:

**XGB2GPU:** Desenvolver uma ferramenta de exploração do espaço de poda de modelos XGBoost baseada em GPU, capaz de avaliar milhares de configurações de poda em paralelo, implementando e comparando diferentes estratégias topológicas (Flat, Linear e Exponencial) para seleção de nós candidatos à remoção;

**RDSF:** Aprimorar o mapeamento de modelos Random Forest em FPGAs através do uso de Binary Decision Diagrams (BDDs) para otimização de funções Booleanas, reduzindo a complexidade dos circuitos gerados;

**TreeLUT:** Estender a ferramenta TreeLUT para incluir implementação alternativa baseada em multiplexadores além da abordagem por equações, contabilização completa dos recursos de hardware incluindo módulos de quantização de entrada e argmax, e exploração do espaço de hiperparâmetros de quantização ($w_{feature}$ e $w_{tree}$) para análise de trade-offs entre acurácia e área;

**Go-Fast:** Desenvolver um simulador de circuitos digitais baseados em LUTs acelerado por GPU, capaz de executar simulações de alto volume (superiores a $2^{30}$ estímulos) com speedups de 5 ordens de magnitude em relação ao Verilator, implementar estratégias de poda aproximada a nível de circuito com avaliação paralela de múltiplas variantes, e suportar diferentes métricas de erro (classificação e regressão ponderada) para validação de circuitos aproximados;

**Validação Experimental:** Avaliar o conjunto de ferramentas desenvolvido utilizando benchmarks estabelecidos (Adult, Covtype, Dry Bean e conjuntos sintéticos), além de circuitos de referência (ISCAS85 e EPFL), analisando os ganhos de velocidade alcançados em relação a implementações baseadas em CPU, trade-offs entre redução de área (LUTs) e degradação de acurácia, comparação com o estado da arte em síntese lógica aproximada, e viabilidade de diferentes níveis de quantização e poda.

## 1.4 Estrutura do Trabalho

A Figura 1 apresenta uma visão geral do fluxo metodológico da dissertação. Partindo de um conjunto de dados original, o processo pode ser dividido em quatro etapas principais:

**Etapa 1 - Pré-processamento e Treinamento:** Aplicam-se técnicas de recodificação, que englobam redução de dimensionalidade e quantização, para criar um novo conjunto de dados otimizado. Sobre os dados originais ou reduzidos, aplicam-se modelos tradicionais de aprendizado de máquina (Random Forest e XGBoost) utilizando bibliotecas do estado da arte (*scikit-learn* e *XGBoost*).

**Etapa 2 - Poda de Modelos:** Em relação à floresta de árvores gerada pelos modelos, aplica-se uma estratégia de poda com exploração massiva do espaço de projeto via GPU (ferramenta XGB2GPU), avaliando milhares de configurações em paralelo para identificar a melhor relação entre redução de nós e preservação de acurácia.

**Etapa 3 - Mapeamento em Hardware:** Para a floresta original ou podada, realiza-se o mapeamento em hardware reconfigurável. Mais especificamente, gera-se código

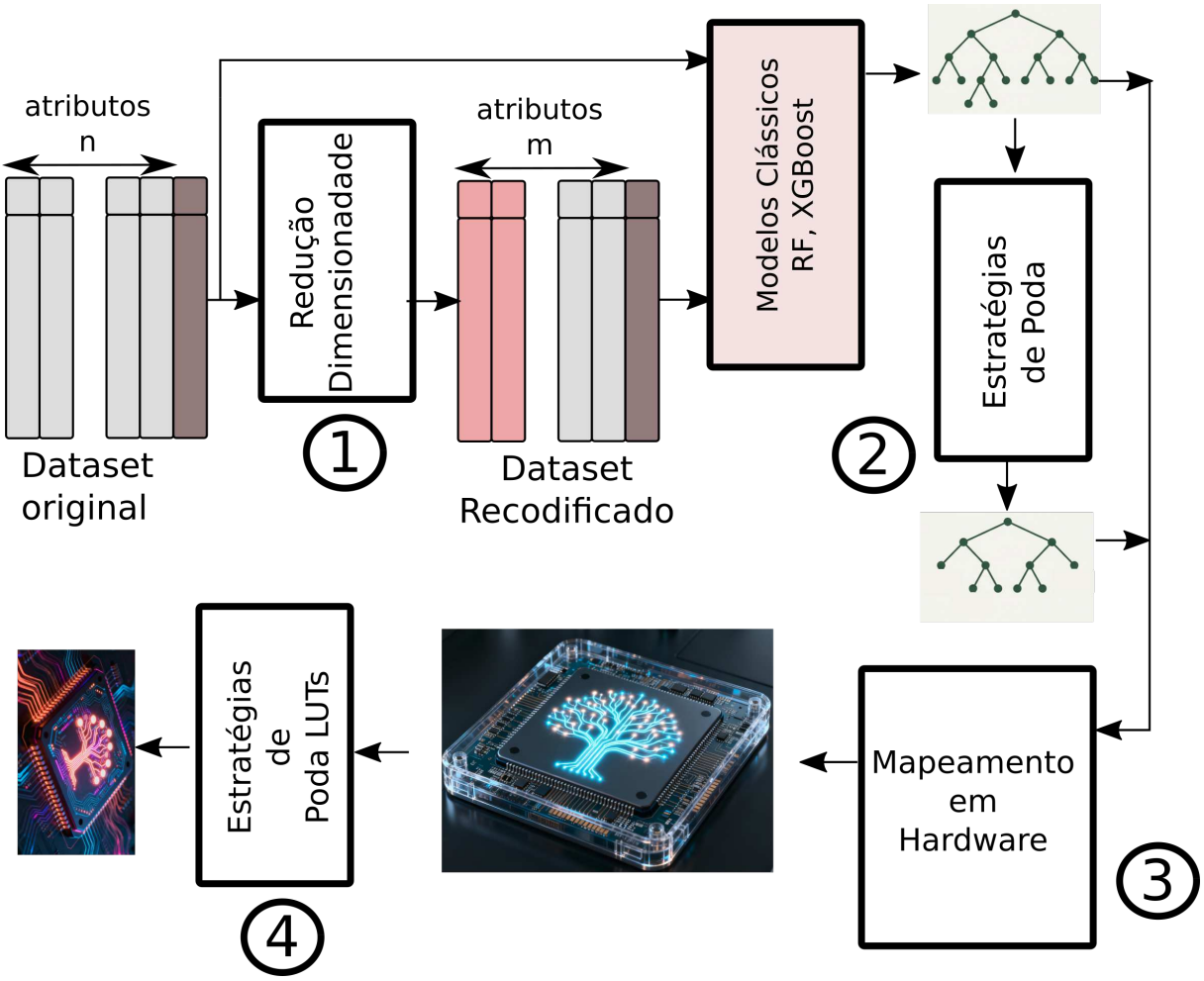



Figura 1 – Diagrama Geral das Ferramentas desenvolvidas na Dissertação

Verilog através das ferramentas RDSF (para Random Forest) ou TreeLUT (para XGBoost), e utilizam-se ferramentas de síntese (Yosys/ABC) para o mapeamento em LUTs do FPGA.

**Etapa 4 - Otimização de Circuito:** Uma vez validado o circuito, exploram-se técnicas de poda a nível de LUTs utilizando a ferramenta Go-Fast, que emprega simulação aproximada acelerada por GPU para avaliar múltiplas variantes de circuito, visando a minimização adicional de recursos de hardware.

Este trabalho está organizado em cinco capítulos, estruturados da seguinte forma:

O **Capítulo 2** dedica-se à fundamentação teórica, revisando conceitos essenciais sobre FPGAs e suas ferramentas de síntese. Ressalta-se a capacidade das técnicas clássicas de aprendizado de máquina de gerarem modelos mapeáveis diretamente em funções Booleanas de múltiplas saídas. Descrevem-se as GPUs e seu modelo de programação CUDA, empregados na exploração paralela do espaço de projetos. Subsequentemente, apresentam-se os algoritmos de aprendizado baseados em árvores de decisão (Random Forest e XGBoost). Para cada algoritmo, discutem-se as possibilidades de mapeamento em representações de hardware, incluindo abordagens baseadas em multiplexadores, equa-

ções e tabelas. Dado o foco deste trabalho na aceleração e implementação de hardware, prioriza-se a etapa de inferência em detrimento do treinamento.

O **Capítulo 3** detalha a metodologia adotada e as quatro ferramentas desenvolvidas para alcançar os objetivos propostos. A ferramenta XGB2GPU realiza pré-mapeamento para poda de modelos XGBoost com exploração paralela em GPU. A ferramenta RDSF efetua o mapeamento de Random Forest para FPGA utilizando BDDs. A ferramenta TreeLUT, estendida neste trabalho, realiza o mapeamento de XGBoost com quantização em múltiplos níveis. Por fim, a ferramenta Go-Fast executa pós-mapeamento para simulação acelerada e poda de circuitos baseados em LUTs.

O **Capítulo 4** apresenta os experimentos realizados e os resultados obtidos, organizados por ferramenta. Para cada ferramenta, descrevem-se os conjuntos de dados utilizados, os parâmetros experimentais, as métricas de avaliação e a análise comparativa dos resultados. Destaca-se a análise de ganho de velocidade alcançados (900× a 3000× para poda de árvores e cinco ordens de magnitude para simulação de circuitos), os trade-offs entre redução de área e degradação de acurácia para diferentes estratégias de poda, o impacto dos hiperparâmetros de quantização ($w_{feature}$ e $w_{tree}$) na acurácia e no uso de LUTs, a comparação com ferramentas do estado da arte (Verilator, ResubALS), e a demonstração da superioridade de simulação com grandes volumes de estímulos.

O **Capítulo 5** sintetiza as principais contribuições da pesquisa, discute as limitações dos métodos propostos, e sugere direções promissoras para trabalhos futuros, incluindo a extensão para outros modelos baseados em árvores, a integração com ferramentas comerciais de FPGA, a exploração de técnicas de poda conscientes de aplicação, e a validação em aplicações reais de TinyML.

# 2 Fundamentos e Conceitos

Neste capítulo serão apresentados os principais conceitos e fundamentos. Primeiramente, será discutida a arquitetura alvo, mais especificamente os FPGAs, que serão utilizados na implementação dos modelos de aprendizado de máquina em hardware. Em seguida, serão apresentados os principais algoritmos de aprendizado de máquina utilizados no estudo, com ênfase nas árvores de decisão. Por fim, serão discutidas as características das GPUs e sua aplicação na aceleração do processo de exploração do espaço de projeto.

## 2.1 Arquitetura Alvo

Nesta seção serão apresentados os principais componentes lógicos digitais das arquiteturas reconfiguráveis, mais especificamente dos FPGAs, utilizados na implementação dos modelos de aprendizado de máquina em hardware. Serão discutidos os conceitos básicos e seu funcionamento.

### 2.1.1 *Field Programmable Gate Arrays* FPGA

*Field-Programmable Gate Arrays* (FPGAs) são dispositivos semicondutores que, diferentemente dos ASICs (*Application-Specific Integrated Circuits*), oferecem a vantagem de serem reconfiguráveis, permitindo a implementação de qualquer lógica digital dedicada a um problema após a fabricação.

A arquitetura básica de um FPGA é composta por uma matriz de blocos lógicos configuráveis, contendo *flip-flops*, LUTs (*Look-Up Tables*) e multiplexadores, interconectados por uma malha reconfigurável (BOUTROS; BETZ, 2021). Neste trabalho, os FPGAs serão utilizados como plataforma de implementação dos modelos de aprendizado de máquina, devido à sua flexibilidade e capacidade de paralelismo, essenciais para aplicações em computação de borda.

### 2.1.2 Look-up Tables

*Look-up Tables*(LUTs) podem ser entendidas como memórias de pequena escala que possuem valores pré-definidos. Esses componentes possuem $n$ entradas, utilizadas para indexar a tabela e obter a saída correspondente de 1 bit. Dessa forma, uma LUT com $n$ entradas pode representar qualquer função booleana de $n$ variáveis, armazenando $2^n$ combinações possíveis de entradas e suas respectivas saídas (FRANCIS, 1992).

A Figura 2 ilustra o mapeamento da função booleana de 3 variáveis $y = (a \wedge b) \vee \not{c}$ para uma LUT de 3 entradas. A tabela central apresenta todas as $2^3$ combinações possíveis

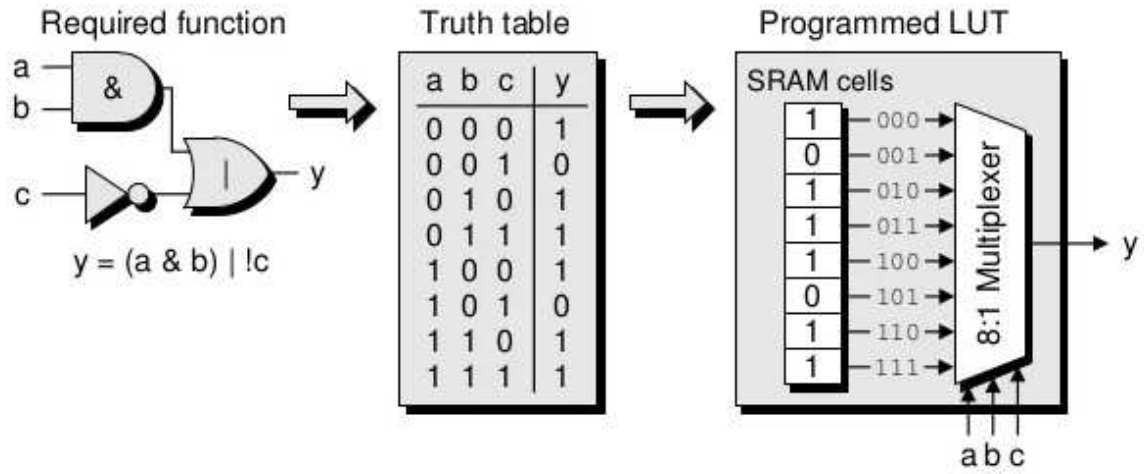



Figura 2 – Diagrama comparativo entre uma LUT de 3 entradas e uma função booleana equivalente (YASPR, ).

das entradas e suas respectivas saídas. A LUT, representada à direita, armazena essas saídas em posições de memória indexadas pelas combinações das entradas. Por exemplo, para a entrada $abc = 101$, a saída correspondente é 1, conforme indicado na tabela.

Neste estudo, as LUTs serão identificadas pelo número de entradas que possuem, portanto, uma LUT de 3 entradas será referenciada como LUT3, uma LUT de 4 entradas como LUT4, e assim sucessivamente. Esse bloco lógico será a métrica principal para avaliar a eficiência das implementações propostas, uma vez que a quantidade de LUTs utilizadas impacta diretamente na área ocupada no *FPGA*.

### 2.1.3 Modelos de Aprendizado são Funções Booleanas?

Esta seção ilustra, por meio de um exemplo simplificado, como um modelo de aprendizado de máquina baseado em dados categóricos tabulares pode ser transformado em uma função Booleana. O cenário proposto consiste na classificação de frutas, conforme esquematizado na Figura 3.

O processo inicia-se com os dados qualitativos (categóricos) das frutas, percorre a etapa de codificação numérica, converte os valores para representação binária e finaliza com a modelagem de uma função Booleana apta para implementação em circuitos digitais.

A Tabela 1 apresenta o conjunto de dados inicial, composto por cinco amostras descritas por atributos qualitativos. O objetivo é classificar a fruta (classe alvo) entre três possibilidades: Melancia, Laranja ou Banana.

Para viabilizar o processamento digital, é necessário converter esses descritores textuais em valores numéricos. Adotou-se uma codificação baseada em uma escala ordinal de intensidade de 2 bits (valores de 0 a 3).

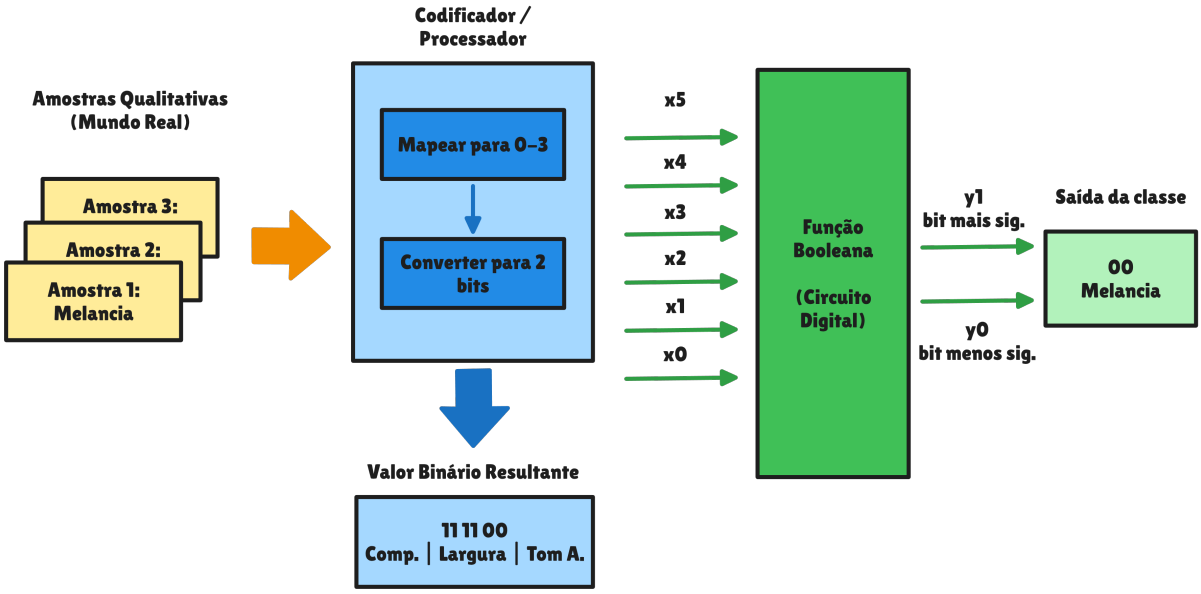



Figura 3 – Fluxo simples para transformar uma tabela categórica em uma função Booleana para um classificador de frutas.

Tabela 1 – Amostras iniciais com atributos qualitativos

| **Comprimento** | **Largura** | **Tom de Amarelo** | **Fruta (Classe)** |
|---|---|---|---|
| Muito Grande | Grande | Inexistente | Melancia |
| Pequeno | Médio | Médio | Laranja |
| Grande | Pequeno | Alto | Banana |
| Médio | Pequeno | Alto | Banana |
| Pequeno | Pequeno | Médio | Laranja |

Nesta abordagem, diferentes adjetivos são agrupados conforme o grau de magnitude que representam. Por exemplo, o valor **0** representa a ausência ou a menor escala do atributo (seja "Inexistente"para cor ou "Muito Pequeno"para dimensões), enquanto o valor **3** representa a magnitude máxima. A Tabela 2 detalha esse esquema de mapeamento unificado.

Tabela 2 – Esquema de codificação: Mapeamento de Texto para Inteiro

| **Atributos (Entrada)** | | **Classes (Saída)** | |
|---|---|---|---|
| **Código** | **Descritores Abrangidos** | **Código** | **Fruta** |
| 0 | Inexistente / Muito Pequeno | 0 | Melancia |
| 1 | Pequeno / Baixo | 1 | Laranja |
| 2 | Médio | 2 | Banana |
| 3 | Grande / Alto / Muito Grande | - | - |

Aplicando-se a regra de conversão da Tabela 2 sobre os dados originais da Tabela 1, obtém-se a representação inteira apresentada na Tabela 3. Note, por exemplo, que na primeira amostra, o atributo "Comprimento: Muito Grande"foi convertido para **3**, e "Tom

de Amarelo: Inexistente"para **0**.

Tabela 3 – Conjunto de dados após a Codificação Numérica (0-3)

| **Comprimento** | **Largura** | **Tom de Amarelo** | **Fruta** |
|---|---|---|---|
| 3 | 3 | 0 | 0 |
| 1 | 2 | 2 | 1 |
| 3 | 1 | 3 | 2 |
| 2 | 1 | 3 | 2 |
| 1 | 1 | 2 | 1 |

Na sequência, converte-se os valores inteiros para o formato binário de 2 bits. Por exemplo, o valor decimal 3 é mapeado para $11_2$, enquanto o valor 1 torna-se $01_2$, resultando na Tabela 4.

Tabela 4 – Dados codificados em Binário

| **Comprimento** (2 bits) | **Largura** (2 bits) | **Tom de Amarelo** (2 bits) | **Fruta** (2 bits) |
|---|---|---|---|
| 11 | 11 | 00 | 00 |
| 01 | 10 | 10 | 01 |
| 11 | 01 | 11 | 10 |
| 10 | 01 | 11 | 10 |
| 01 | 01 | 10 | 01 |

Finalmente, o problema é modelado como uma tabela de uma função Booleana incompleta. O sistema possui 3 atributos de 2 bits cada, totalizando 6 variáveis de entrada ($x_5$ a $x_0$). A saída, correspondente à classe da fruta, é representada por 2 bits ($y_1, y_0$).

- **Entradas:** Comprimento ($x_5x_4$), Largura ($x_3x_2$), Amarelo ($x_1x_0$).
- **Saídas:** Fruta ($y_1y_0$).

Tabela 5 – Tabela Verdade Parcial (Entradas e Saídas Booleanas)

| **Comprimento** | | **Largura** | | **Amarelo** | | **Saída** | |
|---|---|---|---|---|---|---|---|
| $x_5$ | $x_4$ | $x_3$ | $x_2$ | $x_1$ | $x_0$ | $y_1$ | $y_0$ |
| 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 |
| 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 |

Considerando as seis variáveis de entrada, o espaço total de endereçamento é de $2^6 = 64$ combinações. Contudo, a Tabela 5 especifica apenas 5 linhas, restando 59 combinações não observadas ("don't cares"ou casos de teste não vistos).

No contexto desta dissertação, diferentemente deste exemplo ilustrativo, as técnicas de aprendizado de máquina abordadas geram funções Booleanas completamente especificadas. O objetivo central consiste em induzir um modelo capaz de generalizar o conhecimento a partir de um conjunto de dados incompleto, definindo uma resposta válida para qualquer entrada possível. Matematicamente, busca-se encontrar as funções Booleanas $f_1$ e $f_0$ tais que $y_1 = f_1(x_5, \ldots, x_0)$ e $y_0 = f_0(x_5, \ldots, x_0)$, minimizando o erro de classificação.

### 2.1.4 Ferramentas de Síntese

Para este trabalho, as ferramentas desenvolvidas convertem modelos abstratos, tais como árvores de decisão, em código Verilog, visando a representação em *hardware*. Para a etapa seguinte, para o mapeamento do Verilog no FPGA, foram utilizadas duas principais ferramentas: Yosys (WOLF, 2016) e ABC (BRAYTON; MISHCHENKO, 2010), empregadas na síntese e no mapeamento do circuito digital em LUTs. As duas ferramentas operam de forma integrada, sendo o Yosys responsável pela síntese inicial do código HDL (*Hardware Description Language*) descrito em Verilog para uma representação interna de funções Booleanas, enquanto o ABC executa a otimização dessas funções e o mapeamento final para a tecnologia alvo, que neste caso são as LUTs dos FPGAs. Consequentemente, cada célula mapeada restringe-se ao uso local de uma função com apenas $n$ entradas, onde $n$ depende do tamanho da LUT. O circuito resultante constitui, portanto, um grafo de LUTs.

Além de otimizar a lógica, essas ferramentas são fundamentais para reestruturar a topologia do circuito, originalmente composta por múltiplas camadas de variáveis intermediárias. Por exemplo, as entradas primárias $x_0, \ldots, x_n$ alimentam uma camada de $m$ comparadores definidos por $C_i = f(x_0, \ldots, x_n)$, com $0 \leq i < m$. A camada subsequente implementa a lógica das árvores, utilizando como entrada os resultados dos comparadores, tal que $A_i = f(C_0, \ldots, C_m)$. Nesse contexto, a técnica de *flattening* é empregada para expressar as saídas $S$ exclusivamente em função das entradas primárias, resultando em $S_i = f(x_0, \ldots, x_n)$.

## 2.2 *Graphical Processing Units (GPUs)*

Criada em um primeiro contexto de renderização gráfica, as *Graphical Processing Units (GPUs)* evoluíram para se tornarem motores de processamento paralelo eficiente, capazes de lidar com tarefas computacionais intensivas em diversas áreas, incluindo aprendizado de máquina. A arquitetura das GPUs é caracterizada por um grande número de núcleos de processamento *CUDA cores*, que permitem a execução simultânea de milhares de threads, tornando-as ideais para operações que podem ser massivamente paralelizadas.

No entanto, extrair o desempenho máximo dessa arquitetura exige a consideração de restrições de hardware específicas. Embora as GPUs possuam subsistemas de memória de alta vazão, a latência de memória permanece elevada (entre 20 e 30 ciclos de clock) em comparação com CPUs *multi-core* (LUO et al., 2024). Além disso, as operações lógicas, que executam em pipeline, tipicamente levam de 4 a 6 ciclos (ARAFA et al., 2019). Adicionalmente, o modelo de execução SIMT (*Single Instruction Multiple Thread*) impõe desafios para simulações orientadas a eventos, visto que a divergência de desvios pode degradar severamente o desempenho. Portanto, implementações eficientes devem minimizar seções de código com muitos desvios e garantir uma razão ótima entre threads e unidades de computação (geralmente entre 4 e 10) para manter a utilização plena do hardware.

Considerando essas características, neste trabalho será usada uma estratégia de escalonamento de recursos em GPU baseada em ***More Work per Thread***(VOLKOV, 2010). Nela, cada thread é responsável por processar múltiplos elementos de dados, reduzindo a sobrecarga de gerenciamento de threads e aumentando a eficiência do uso da memória compartilhada. Portanto, fazendo com que cada thread realize um ciclo completo de trabalho, não o particionando, mas sim atribuindo a ela a responsabilidade por um conjunto maior de dados, é possível explorar o paralelismo em nível de instrução (ILP) e mitigar a latência de memória. Essa abordagem é particularmente benéfica para operações que envolvem acesso frequente à memória, como as encontradas em algoritmos de aprendizado de máquina, e quando é feita uma exploração de um espaço amostral maior.

Neste trabalho, a GPU foi empregada para explorar o espaço de projeto de otimizações baseadas na poda de árvores. A estratégia consiste na avaliação paralela de uma configuração de poda distinta por cada *thread*. Dado que o acesso à memória representa o principal gargalo de desempenho, a eficiência é maximizada através do compartilhamento dos dados de entrada entre as *threads*. Adicionalmente, esse paradigma foi estendido à poda de grafos de LUTs em FPGAs, mantendo o princípio de avaliação independente de podas por *thread*.

## 2.3 Algoritmos de Aprendizado de Máquina

Nesta seção serão apresentados os principais algoritmos de aprendizado de máquina utilizados no estudo. Além dos seus funcionamentos básicos, serão discutidas suas vantagens e desvantagens em diferentes contextos.

O aprendizado de máquina fundamenta-se na construção de algoritmos capazes de aprender padrões a partir de dados. Formalmente, define-se um espaço de entrada como $\mathcal{X} \subseteq \mathbb{R}^M$, constituído de $M$ elementos que caracterizam a amostra, e um espaço de saída como $\mathcal{Y}$. O objetivo central é estimar uma função desconhecida $f : \mathcal{X} \to \mathcal{Y}$ que mapeia as variáveis de entrada para as variáveis alvo, de modo a generalizar o aprendizado para

dados não observados.

Para realizar tal estimativa, utiliza-se um conjunto de dados de treinamento, que neste trabalho será definido como $D = \{(x_i, y_i)\}_{i=1}^{n}$. A representação algébrica deste conjunto é dada pela matriz $X \in \mathbb{R}^{n \times M}$, onde $M$ denota a dimensionalidade do vetor de **atributos**. A matriz $X$ é estruturada tal que cada linha representa uma amostra e cada coluna uma variável de predição:

$$X = \begin{bmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,M} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,M} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n,1} & x_{n,2} & \cdots & x_{n,M} \end{bmatrix} \tag{2.1}$$

onde $x_{i,j}$ corresponde ao valor da $j$-ésimo atributo ou atributo da $i$-ésima amostra. O vetor alvo é denotado por $\mathbf{y} \in \mathcal{Y}^n$. No contexto específico de classificação, $\mathbf{y} = [y_1, y_2, \ldots, y_n]^T$ assume valores discretos pertencentes a um conjunto finito de classes $\mathcal{C} = \{c_1, \ldots, c_k\}$. Dessa forma, o par ordenado $(x_i, y_i)$ encapsula a relação supervisionada entre os atributos observados e a classe real que o modelo busca prever.

Além dos dados de treinamento, existem os dados de teste, obscuros ao modelo durante a fase de treino e que são responsáveis por avaliar a capacidade de generalização do modelo. E os dados de validação, que podem ser utilizados para ajustar hiperparâmetros e evitar o sobreajuste, neste trabalho eles serão utilizados para avaliar a acurácia dos modelos podados. Como já mencionado, o foco deste trabalho é a implementação do processo de inferência, sempre tendo o cuidado de realizar qualquer ajuste guiado apenas pelo conjunto de treino, à verificação final é realizada com os dados de testes.

### 2.3.1 Árvore de Decisão

Matematicamente, uma árvore de decisão $T$ pode ser definida como um grafo acíclico dirigido $G = (V, E)$, onde $V$ representa o conjunto finito de nós e $E$ o conjunto de arestas que definem a hierarquia da estrutura. O conjunto de nós $V$ é particionado em dois subconjuntos disjuntos: o conjunto de nós de decisão (ou internos), denotados por $\mathcal{N}$, e o conjunto de nós folha (ou terminais), denotados por $\mathcal{F}$, tal que $V = \mathcal{N} \cup \mathcal{F}$ e $\mathcal{N} \cap \mathcal{F} = \emptyset$.

A estrutura topológica impõe que exista um único nó raiz $r \in \mathcal{N}$ com grau de entrada zero. Todo nó $v \in V \setminus \{r\}$ possui grau de entrada igual a 1, garantindo que haja um único caminho entre a raiz e qualquer nó. Para árvores binárias, cada nó interno $v \in \mathcal{N}$ possui grau de saída igual a 2, conectando-se a um nó filho à esquerda, $child_L(v)$, e a um nó filho à direita, $child_R(v)$.

O espaço de entrada $x \subseteq \mathbb{R}^M$, onde $M$ é a dimensionalidade dos dados, é particio-

nado hierarquicamente pela árvore. A cada nó interno $v \in \mathcal{N}$ está associada uma função de divisão $c_v : x \to \{\text{esquerda}, \text{direita}\}$, parametrizada por uma tupla $\theta_v = (j, t_m)$, onde $j \in \{1, \ldots, M\}$ representa o índice da característica avaliada e $t_m \in \mathbb{R}$ o limiar de corte (*threshold*). A regra de decisão em $v$ para um vetor de entrada $\mathbf{x} = (x_1, \ldots, x_M)$ é dada por:

$$c_v(\mathbf{x}) = \begin{cases} child_L(v) & \text{se } x_{j_v} > t_v \\ child_R(v) & \text{se } x_{j_v} \leq t_v \end{cases} \tag{2.2}$$

Os nós folha $f \in \mathcal{F}$ representam as regiões finais da partição do espaço e não realizam comparações. A cada folha $f$ é atribuído um valor de saída $w_f$, que corresponde à predição do modelo para as amostras que alcançam tal nó. Desta forma, a inferência de uma árvore de decisão pode ser vista como uma função composta que roteia a entrada $\mathbf{x}$ da raiz até uma folha única $f^*$, retornando $w_{f^*}$.

Dentre os hiperparâmetros estruturais de uma árvore de decisão, a profundidade máxima, denotada por $Depth_{max}$, estabelece o limite superior para a distância entre a raiz e qualquer folha. Embora não seja o único parâmetro regulador e nem necessariamente o principal critério para ditar a qualidade preditiva do modelo, o foco nesta métrica justifica-se por seu impacto direto no consumo de recursos e na latência do hardware gerado. O controle estrito desse parâmetro é crucial porque o crescimento da árvore é exponencial: ao incrementar a profundidade de $l$ para $l + 1$, o número máximo de nós pode dobrar, tal que o total de folhas possíveis é limitado por $2^{D_{max}}$ e o número total de nós no modelo é limitado por $\sum_{i=0}^{D_{max}} 2^i = 2^{D_{max}+1} - 1$.

Devido ao crescimento exponencial, o ajuste de $D_{max}$ e as técnicas de poda influenciam diretamente o tamanho da árvore. Em geral, os algoritmos de treinamento usam uma técnica gulosa para escolher as variáveis de decisão em cada nó. Para problemas de classificação, as técnicas mais comuns baseiam-se nos coeficientes de entropia ou Gini, que medem o grau de pureza dos dados na divisão gerada pelo nó. Uma boa decisão ocorre quando o atributo escolhido e seu valor de comparação dividem bem as classes naquele ramo (separando, por exemplo, "bananas para um lado e melancias para o outro"). Outro hiperparâmetro é o número mínimo de amostras, utilizado para evitar a subdivisão de conjuntos pequenos, o que reduziria a capacidade de generalização. Este trabalho foca na etapa de inferência, isto é, com a árvore já construída, visa-se mapeá-la de forma eficiente no *hardware*. Árvores profundas possuem um grande espaço de hipóteses, capaz de capturar detalhes granulares dos dados de treinamento, o que reduz o viés, mas aumenta significativamente a propensão ao sobreajuste. Em contrapartida, árvores rasas limitam a complexidade computacional e de memória, mas podem sofrer de subajuste ao falharem na captura de padrões não-lineares complexos.

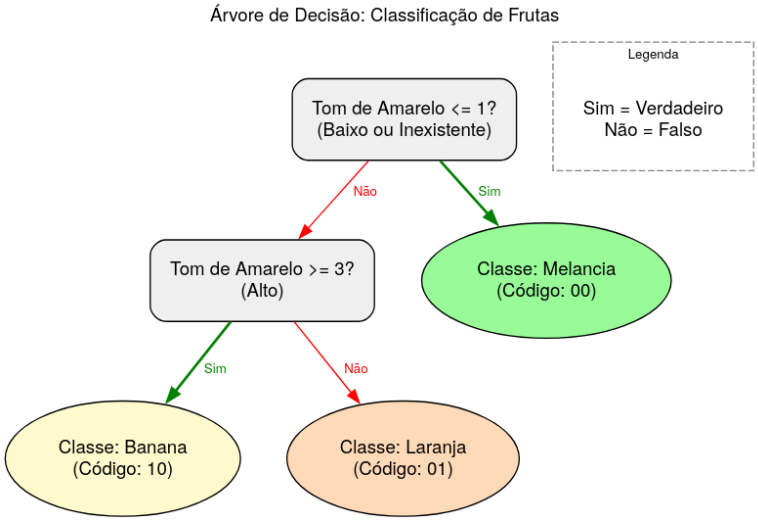



Figura 4 – Exemplo de uma Árvore de Decisão para Classificação de Frutas

A Figura 4 exemplifica uma árvore de decisão aplicada à classificação de frutas. Neste cenário, os nós de decisão baseiam-se exclusivamente no atributo de tom de amarelo. O nó raiz isola as melancias das demais frutas, resultando em uma folha de classe "pura". O ramo remanescente agrupa bananas e laranjas, onde uma nova avaliação do tom, sob um limiar distinto, mostra-se suficiente para discriminar as duas classes. Destaca-se a interpretabilidade inerente a este modelo, propriedade que facilita a explicação das decisões tomadas. Em suma, este exemplo simples demonstra que a cor constitui um atributo eficaz para realizar uma classificação satisfatória.

O próximo passo consiste em transformar a árvore de decisão em um circuito lógico, visando sua implementação em FPGA.

### 2.3.2 Árvore de Decisão em Hardware

Para que o modelo de aprendizado de máquina baseado em árvores de decisão possa ser utilizado em um contexto de computação em borda, é essencial que essas estruturas sejam implementadas em *hardware*. Nesse cenário, diversas abordagens de implementação se apresentam como alternativas viáveis. Uma alternativa direta é usar comandos condicionais para os nós implementados com um conjunto de multiplexadores (IKEDA et al., 2020; SILVA et al., 2024).

A Figura 5(A) apresenta uma árvore de decisão com três classes (vermelho, azul e verde) e dois nós internos (C1 e C2), que correspondem às comparações realizadas para classificar uma amostra, neste caso, na classe verde. As folhas tem 3 bits para implementar com one-hot code cada uma das classes 001 para verde, 010 para azul e 100 para vermelho. Por sua vez, a Figura 5(B) ilustra a mesma árvore representada em termos de multiplexadores, onde cada nó interno é substituído por um multiplexador de 2 para 1. As entradas dos multiplexadores são determinadas pelas classes associadas às folhas


Figura 5 – (A) Exemplo de árvore de decisão (B) Representação usando multiplexadores.

da árvore, enquanto os sinais de controle são derivados das comparações realizadas nos nós internos. Por exemplo, o multiplexador correspondente ao nó C2 seleciona entre as classes azul e verde com base no resultado da comparação $x_3 < 4$. A saída final do último multiplexador representa a predição da árvore de decisão para a amostra de entrada. Importante destacar que os multiplexadores e os fios irão carregar uma informação de 3 bits de largura.

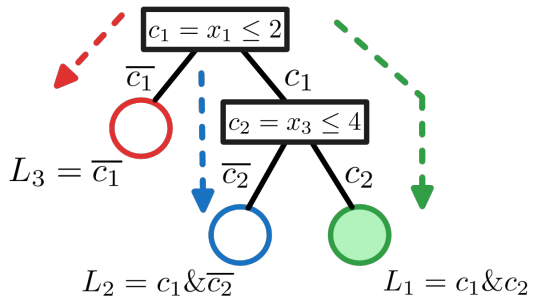
O algoritmo 1 apresenta a implementação em pseudocódigo da árvore de decisão ilustrada na Figura 5, utilizando a notação inspirada em linguagens de descrição de hardware, como Verilog e VHDL. O algoritmo é dividido em dois estágios principais: o estágio de comparação e o estágio de multiplexação. No primeiro estágio, são gerados os sinais de controle ($c\_1$ e $c\_2$) através das comparações dos atributos de entrada ($X_1$ e $X_3$) com os limiares definidos na árvore. No segundo estágio, os multiplexadores são utilizados para propagar a decisão da árvore de baixo para cima, selecionando as classes apropriadas com base nos sinais de controle gerados anteriormente. A saída final $Y$ representa a classe predita pela árvore de decisão para a amostra de entrada.

**Algoritmo 1:** Árvore de Decisão em Hardware (Lógica de Multiplexadores)

```
1 /* 1. Estágio de Comparação (Geração dos Sinais de Seleção)    */
2 wire c_1 ← (X_1 ≤ 2)
3 wire c_2 ← (X_3 ≤ 4)
4 /* 2. Estágio de Multiplexação (Propagação Bottom-Up)          */
5 assign mux_2_out = c_2?L_1 : L_2
6 assign tree_out = c_1?mux_2_out : L_3
7 assign Y ← tree_out
```

Outro método para representar uma árvore de decisão em hardware consiste na utilização equações booleanas, que usam expressões lógicas para representar explicitamente o caminho de travessia de uma amostra pela árvore. A Figura 6(A) mostra a mesma árvore de decisão apresentada anteriormente, enquanto o Algoritmo 2 detalha a implementação em pseudocódigo utilizando essa abordagem. No algoritmo, cada condição

de decisão é representada por uma variável booleana ($c\_1$ e $c\_2$), que indica o resultado das comparações realizadas nos nós internos da árvore. As expressões lógicas ($L_1$, $L_2$, e $L_3$) são então construídas combinando essas variáveis para refletir os caminhos possíveis na árvore. O circuito apresenta três saídas, correspondendo às classes $L_1, L_2$ e $L_3$. Neste exemplo ilustrativo, a abordagem baseada em equações resulta, inicialmente, em uma solução mais compacta. Entretanto, mediante a otimização do circuito de multiplexadores, ferramentas de síntese como o ABC são capazes de simplificar a lógica, alcançando uma eficiência equivalente àquela obtida via equações.




**Algoritmo 2:** Árvore de Decisão em Hardware (Lógica de Equação)

```
1 wire c_1 ← (X_1 ≤ 2)
2 wire c_2 ← (X_3 ≤ 4)
3 assign L_1 = c_1&c_2
4 assign L_2 = c_1&¬c_2
5 assign L_3 = ¬c_1
```

Figura 6 – (A) Exemplo de árvore de decisão com equações (B) Algoritmos da implementação com equações.

Essa representação se trata de uma representação em camadas, na qual os comparadores são definidos como funções das entradas primárias $X$, isto é, $C = f(X)$. A camada subsequente compreende as saídas $L$, expressas em função de $C$ pela relação $L = g(C) = g(f(X))$. O processo de *flattening* consiste, portanto, em representar o sistema integralmente em função de $X$.

Uma ultima estratégia de implementação é utilizando uma memória para armazenar a árvore de decisão em formato de tabela (OWAIDA et al., 2017), com isso é possível indexar os nós das árvores, juntamente com as informações de comparação e classes. A Figura 7(A) apresenta a mesma árvore de decisão, implementada em tabela, com cinco linhas (uma por nó) e cinco colunas, sendo elas a de tipo de nó, interno ou folha, o atributo que cada nó interno usa em sua comparação, o limiar ou classe, a depender do tipo de nó, o filho da esquerda (F1) e o da direita (F2).

A Figura 7(B) ilustra uma variação dessa abordagem, onde é adicionada uma camada de comparadores antes da tabela. Essa camada é responsável por realizar as comparações entre os atributos de entrada e os limiares armazenados na tabela, gerando sinais de controle que são utilizados para indexar a tabela e determinar o próximo nó a ser acessado. Essa modificação pode melhorar a eficiência da implementação, reduzindo o número de acessos à tabela de dados, acelerando a inferência (SAQIB et al., 2015).

A representação em tabela é mais genérica e amplamente utilizada devido à sua flexibilidade de reprogramação. Em contrapartida, implementações baseadas em equações

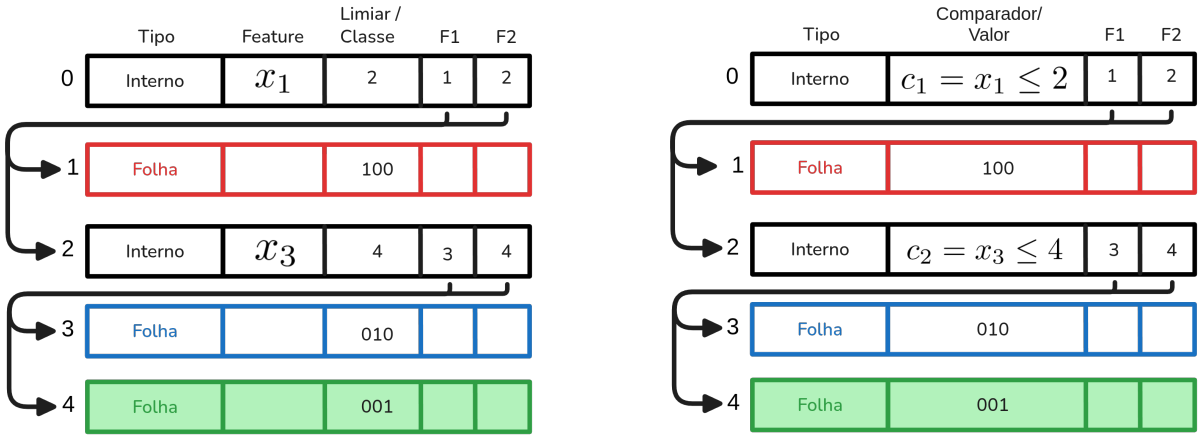



Figura 7 – (A) Exemplo de árvore de decisão com tabela sem camada de comparador (B) Exemplo de árvore de decisão com tabela com camada de comparador.

ou árvores de multiplexadores exigem que o circuito seja regenerado (ou ressintetizado) sempre que ocorrerem alterações na estrutura da árvore.

### 2.3.3 Random Forest

A *Random Forest* (BREIMAN, 2001) é um método de aprendizado *ensemble* composto por uma coleção de árvores de decisão. Formalmente, dado um conjunto de dados de treinamento $D$, o algoritmo treina $B$ árvores independentes. Cada árvore $T_b$ é construída utilizando uma amostra $D_b$ selecionada de $D$ com reposição (técnica conhecida como *bagging* (BREIMAN, 1996)). Para promover a diversidade e descorrelacionar as árvores, em cada nó de divisão, apenas um subconjunto aleatório de atributos de tamanho $m < M$ é considerado candidata para a determinação do melhor ponto de corte.

O processo de treinamento visa particionar o espaço de características recursivamente. Para um nó contendo um subconjunto de dados $Q$, busca-se um parâmetro de divisão $\theta = (j, t_m)$ para construir uma árvore de decisão, onde $j$ é o atributo e $t_m$ é o limiar (*threshold*). O objetivo é encontrar o par $\theta$ que minimize a função de custo baseada na impureza dos nós filhos resultantes, $Q_{esq}$ e $Q_{dir}$. A função de custo $G(Q, \theta)$ é geralmente definida como a soma ponderada da impureza:

$$G(Q, \theta) = \frac{n_{esq}}{n_{total}} H(Q_{esq}) + \frac{n_{dir}}{n_{total}} H(Q_{dir}) \tag{2.3}$$

onde $H(\cdot)$ é uma métrica de impureza, como o índice Gini (MENZE et al., 2009), definido por $H(Q_m) = \sum_k p_{mk}(1 - p_{mk})$, sendo $p_{mk}$ a proporção de amostras da classe $k$ no nó $m$. A divisão ótima $\theta^*$ é aquela que minimiza $G$, resultando na maximização do ganho de informação. Esse processo é repetido até que um critério de parada seja atingido, como uma profundidade máxima ou um número mínimo de amostras por folha.

No processo de inferência, as predições individuais de cada árvore são agregadas.

Para classificação, utiliza-se o *majority vote*, onde a classe final $\hat{y}$ é determinada pela moda das saídas das árvores: $\hat{y} = \text{mode}\{T_1(x), \ldots, T_B(x)\}$. A Figura 8 ilustra esse processo. Neste exemplo, a classe azul vence por receber dois votos de um total de três árvores. Cada nó interno, representado em preto, corresponde à operação $j < t_m$ otimizada durante o treinamento, aqueles que estão preenchidos simbolizam o caminho percorrido pela amostra de entrada até chegar à folha, que contém a predição da árvore.

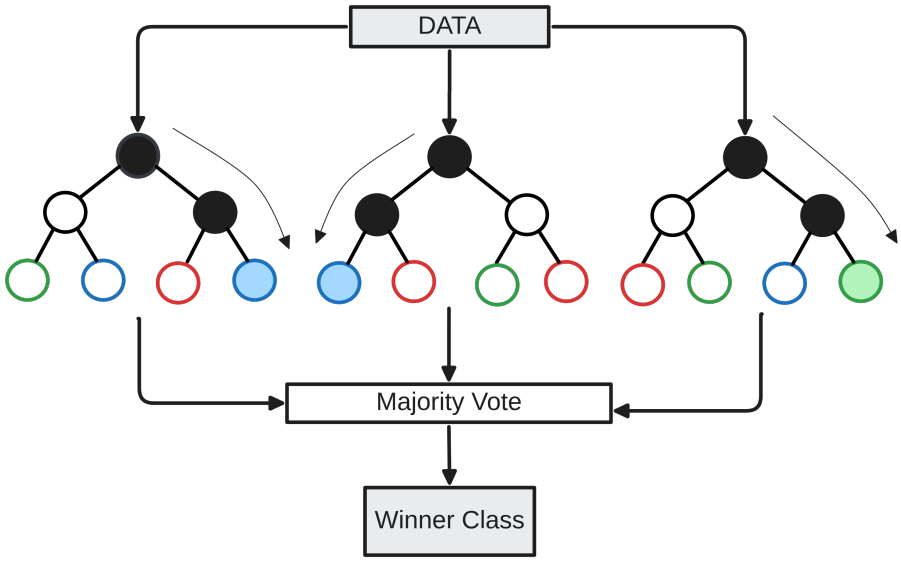



Figura 8 – Processo de inferência de uma *Random Forest*

As implementações de *Random Forest* dois hiper-parâmetros principais: **o número de árvores $B$ e a profundidade máxima das árvores** $Depth_{max}$. Esses parâmetros influenciam diretamente o desempenho do modelo, onde um aumento em $B$ tende a melhorar a robustez e reduzir o sobreajuste, enquanto $Depth_{max}$ controla a complexidade individual das árvores, além de outros hiper-parâmetros com máximo de folhas, mínimo de amostras para aplicar uma divisão, dentre outros.

A implementação em *hardware* do modelo de *Random Forest* é feita através dos módulos de das árvores de decisão, podendo assumir qualquer estratégia, soma dos votos e *majority vote*. Um circuito somador, cuja largura de bits é dimensionada em função do número de árvores, implementa a etapa de soma para cada classe. No exemplo de três classes (verde, azul e vermelho), empregam-se três somadores: $S_{azul}, S_{verde}$ e $S_{vermelho}$. Subsequentemente, um circuito de decisão majoritária processa os resultados em cascata: inicialmente, calculam-se os valores intermediários $votos_1, vencedor_1 = (S_{azul} > S_{verde}?S_{azul}, azul : S_{verde}, verde)$, seguidos pela comparação final com a classe vermelha, dada por $Y = (S_{vermelho} > votos_1?vermelho : vencedor_1)$.

### 2.3.4 XGboost

O segundo modelo utilizado neste trabalho também se baseia em árvores de decisão, porém adota a abordagem de *Gradient Boosting* (NATEKIN; KNOLL, 2013). O *XGBoost* (CHEN; GUESTRIN, 2016) constrói um classificador robusto através da combinação sequencial de árvores de regressão. Diferente da *Random Forest*, onde as árvores são independentes, no XGBoost cada nova função é treinada para corrigir os erros de predição acumulados nas iterações anteriores.

Formalmente, dado o conjunto de dados $D = \{(x_i, y_i)\}_{i=1}^{n}$, onde $x_i \in \mathbb{R}^M$ e $y_i$ é o rótulo da classe, o modelo final é representado por uma soma ponderada de $K$ funções:

$$\hat{z}_i = \sum_{k=1}^{K} f_k(x_i), \quad f_k \in \mathcal{F} \tag{2.4}$$

onde $\mathcal{F}$ é o espaço de árvores de regressão e $\hat{z}_i$ representa o valor bruto da predição. Para obter a classificação probabilística, aplica-se a função Sigmoide sobre a soma: $P(y_i = 1|x_i) = \sigma(\hat{z}_i) = (1 + e^{-\hat{z}_i})^{-1}$.

O treinamento busca minimizar uma função objetivo regularizada $\mathcal{L}$, composta pela função de perda $l$ (neste caso, *Log-Loss*) e um termo de penalização $\Omega$:

$$\mathcal{L} = \sum_{i=1}^{n} l(y_i, \hat{z}_i) + \sum_{k=1}^{K} \Omega(f_k) \tag{2.5}$$

O termo $\Omega(f_k)$ penaliza a complexidade da árvore (número de folhas e magnitude dos pesos) para evitar sobreajuste

A construção é iterativa. Na $t$-ésima iteração, calcula-se o gradiente e a hessiana da perda para cada amostra $x_i$, indicando a direção necessária para ajustar a probabilidade. Após determinar a árvore ótima $f_t$ que minimiza a perda, o modelo é atualizado utilizando uma taxa de aprendizado (*learning rate*) $\eta \in (0, 1]$:

$$\hat{z}_i^{(t)} = \hat{z}_i^{(t-1)} + \eta f_t(x_i) \tag{2.6}$$

O parâmetro $\eta$ reduz a contribuição individual de cada nova árvore, obrigando o modelo a aprender de forma mais gradual e generalizável.

Na Figura 9, observa-se o processo de inferência do XGBoost, onde múltiplas árvores de decisão (representadas como conjuntos para as classes 0, 1 e 2) são utilizadas. Para cada classe, várias árvores independentes são treinadas, onde os nós internos (pretos e brancos) representam comparações baseadas nas características de entrada, e os nós folha (coloridos em azul, verde e vermelho) contêm as pontuações de predição. Durante a inferência, uma amostra de entrada percorre todas as árvores de cada classe, acumulando pontuações nos nós folha correspondentes. Essas pontuações são então somadas (repre-

sentado pelo símbolo +) para cada classe, gerando um vetor de pontuação final. A função $\text{argmax}_{x \in X} f(x) = \{x \in X \mid f(x) \geq f(y), \forall y \in X\}$ é então aplicada para determinar a classe vencedora, selecionando aquela com a maior pontuação acumulada.

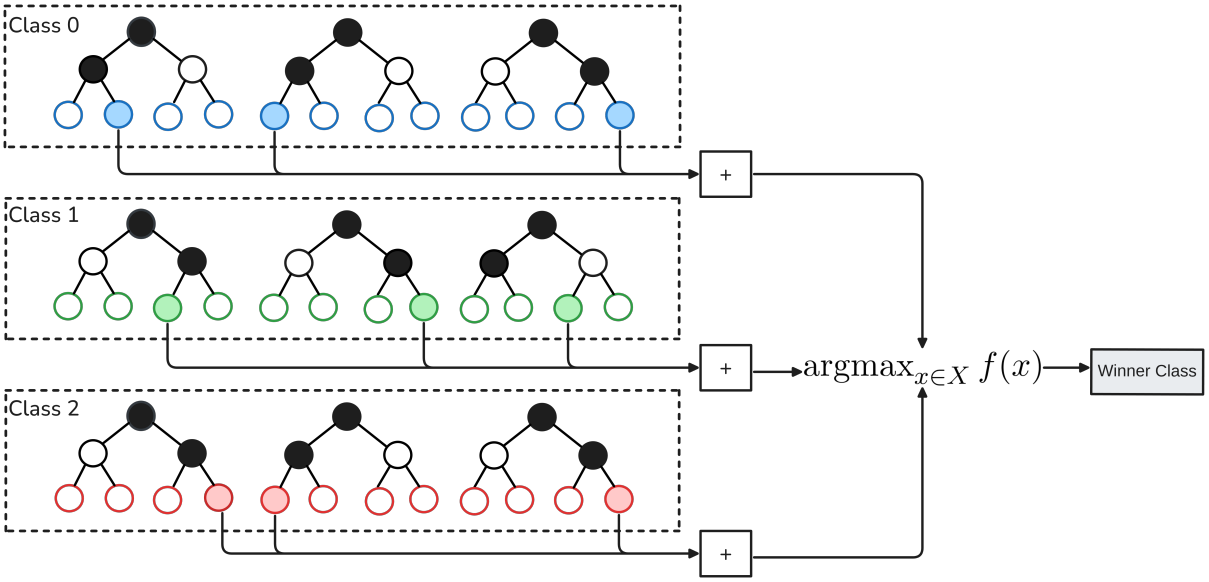



Figura 9 – Processo de inferência de um *XGBoost*

O número de árvores $B$ e a profundidade máxima das árvores $Depth_{max}$ também são os principais hiperparâmetros do XGBoost, com a diferença que, devido à natureza do *boosting*, o número de árvores é multiplicado por classe, ou seja, para um problema de classificação com $k$ classes, o número total de árvores será $B \times k$, enquanto na regressão permanece similar à *Random Forest*, utilizando a soma da pontuação de todas as árvores como previsão final.

A implementação em hardware do XGBoost pode ser realizada utilizando uma abordagem semelhante à da *Random Forest*, empregando multiplexadores para representar as árvores de decisão, visto que cada árvore computa um valor numérico de $n$ bits, a representação tabular apresenta-se como uma alternativa viável. Adicionalmente, a avaliação bit a bit permite o uso de equações ou a geração de uma expressão lógica para a validação de cada folha. Subsequentemente, um mecanismo dedicado acumula as pontuações das árvores de cada classe, enquanto a função argmax, situada na última camada, determina a classe vencedora.

# 3 Ferramentas Implementadas

A metodologia proposta neste trabalho consiste na implementação de ferramentas que atuam em três contextos distintos do fluxo de aprendizado de máquina: (i) poda após a execução do treinamento, etapa aqui denominada *pré-mapeamento*; (ii) ferramentas de mapeamento que convertem os modelos para *hardware*; e (iii) ferramentas de pós-mapeamento, focadas na redução adicional de recursos no FPGA. A organização dessas ferramentas segue uma ordem sequencial. O primeiro grupo é responsável por processar um modelo de entrada (como *XGBoost* ou *Random Forest*) e aplicar uma sequência de otimizações de poda. Nesse contexto, a Seção 3.1 detalha a ferramenta *XGB2GPU*, desenvolvida com base na exploração do espaço de poda de modelos *XGBoost* em GPUs.

As Seções 3.2 e 3.3 apresentam ferramentas de mapeamento de modelos, processo que consiste na conversão do modelo original em código Verilog, tornando-o apto para a síntese em *hardware*. A primeira delas, RDSF, realiza o mapeamento de modelos *Random Forest* para FPGAs utilizando *Binary Decision Diagrams* (BDDs). O BDD constitui uma representação canônica de funções Booleanas; embora apresente crescimento exponencial para certas classes de funções, é capaz de representar outras estruturas de forma compacta e ordenada. A segunda ferramenta, TreeLUT, originalmente apresentada em (KHATAEI; BAZARGAN, 2025), foi estendida visando aprimorar a qualidade das soluções, o que incluiu a reescrita do gerador de código Verilog. O TreeLUT mapeia modelos *XGBoost* em FPGAs aplicando estratégias de quantização em duas etapas: (a) na codificação das entradas; e (b) nos pesos das folhas.

Já o último grupo de ferramentas, agentes de pós-mapeamento, é responsável por otimizar os circuitos digitais gerados pelas ferramentas de mapeamento. A subseção 3.4 apresenta a ferramenta Go-Fast, elaborada durante o desenvolvimento deste trabalho, cujo objetivo é otimizar a simulação aproximada de circuitos digitais baseados em LUTs em *FPGAs*, utilizando *GPUs* para acelerar o processo de simulação, além de aplicar estratégias de poda para reduzir a área ocupada no *FPGA*.

## 3.1 XGB2GPU

Esta seção apresenta uma ferramenta automática de geração de código para GPU, projetada para a exploração do espaço de poda de modelos *XGBoost* na etapa de pós-treinamento. Ao aproveitar a capacidade de processamento paralelo das *GPUs*, o sistema objetiva explorar a concorrência na escala de milhares de *threads* para avaliar múltiplas configurações de poda, buscando minimizar o impacto na precisão preditiva. A seguir, são detalhados os principais componentes e funcionalidades da solução proposta.

A partir do conjunto de dados de entrada, emprega-se a biblioteca *XGBoost* para realizar o treinamento do modelo e gerar o conjunto de árvores. Para entender o funcionamento da ferramenta, é importante definir alguns parâmetros referentes à topologia e modelagem das árvores de decisão. Cada árvore gerada possui $l$ níveis de profundidade, definido como o número de arestas percorridas do nó raiz até um nó $v$. O nível máximo de profundidade de uma árvore é representado por $l_{max}$. O total de nós em uma árvore é denotado por $N_t$, e o total de árvores na floresta é representado por $T$. A ferramenta visa podar um número específico de nós, definido pela meta de redução $R_{cut}$, que representa a quantidade total de nós a serem removidos do modelo na totalidade. Cada thread da GPU é responsável por processar uma configuração de poda, nesse sentido, o total de threads $N_{threads}$ define o número de variantes de poda com um mesmo $R_{cut}$ que serão avaliadas em paralelo.

A Figura 10 ilustra o processo de poda aplicado a uma única árvore de decisão, tendo $R_{cut} = 6$ e duas *threads*. O nó $v$ na profundidade $l$ é avaliado para possível remoção, onde, se a decisão for positiva, o nó interno de comparação, em preto, é substituído por um nó folha com uma pontuação definida por $v_p = \frac{v_{c1}+v_{c2}}{2}$, onde $v_{c1}$ e $v_{c2}$ são os valores das folhas filhas do nó $v$. Na primeira configuração, são escolhidos três nós do nível $l = 2$, resultando na poda de 6 nós folhas. Já na segunda configuração ou segunda *thread*, é escolhido um nó do nível $l = 1$, o que faz com que 4 nós folhas e outros 2 nós, do nível $l = 2$, que propagaram os novos valores de folha, sejam removidos. Os níveis $l = 0$ e $l = l_{max}$ não são considerados para poda, visto que o nó raiz é essencial para a estrutura da árvore e os nós folhas não possuem filhos para serem removidos.

Esse processo é, então, expandido para toda a floresta de árvores e realizado antes da geração do código em CUDA. Sendo assim, a ferramenta gera um vetor de probabilidades de poda. A inferência é realizada para um conjunto de dados de validação, permitindo a avaliação da precisão preditiva de cada variante podada. Com base nos resultados obtidos, é possível selecionar a configuração que melhor equilibra a redução de custo com a remoção de nós sem prejudicar a acurácia do modelo, pois para cada poda todo o conjunto de treino é avaliado.

Dado que a estrutura das árvores e da floresta confere graus distintos de importância aos nós, avaliam-se estratégias de poda alinhadas a essas propriedades. No nível da árvore, a remoção de nós próximos à raiz tende a causar maior impacto na acurácia. Analogamente, em modelos baseados em *boosting*, as árvores iniciais possuem maior peso na decisão final; portanto, a poda nessas estruturas resulta em alterações mais significativas no desempenho do modelo.

A decisão de podar um nó $v$ na profundidade $l$ é definida por uma estratégia de Otimização Global Estocástica, implementada para maximizar a entropia e a diversidade dos modelos gerados pelas threads paralelas. Diferente de abordagens sequenciais que

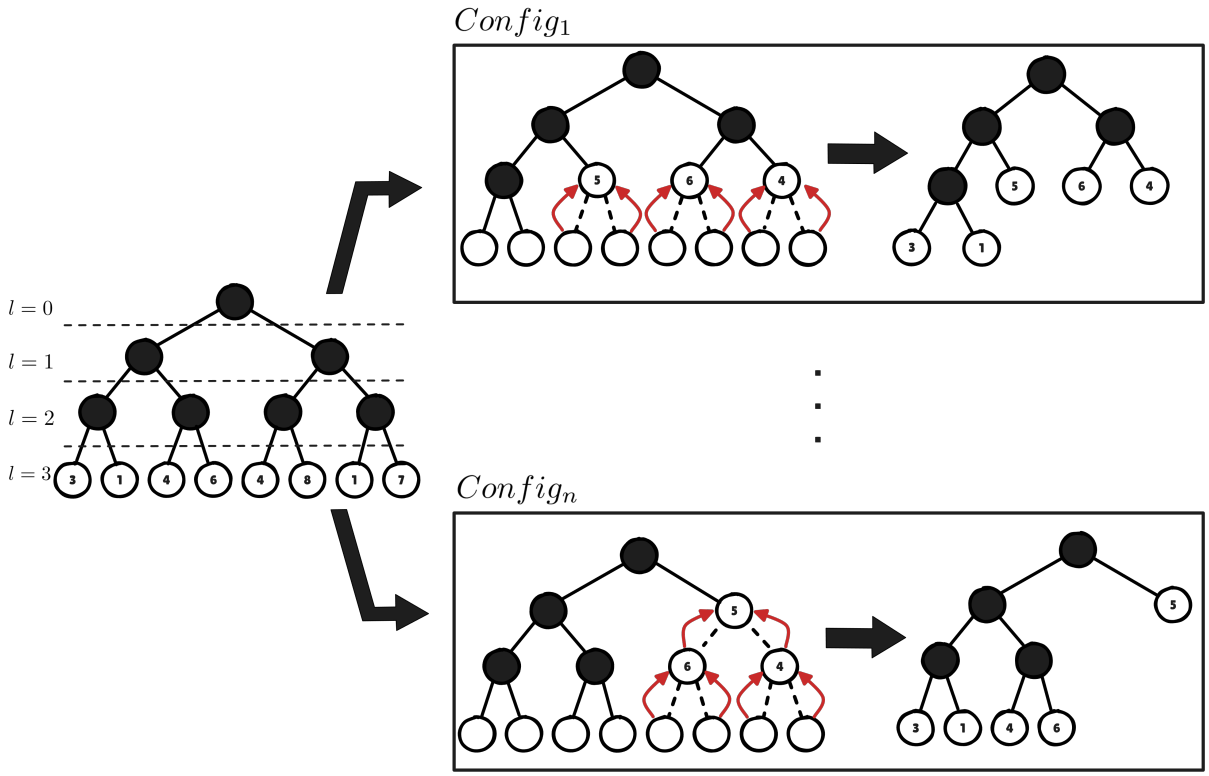



Figura 10 – Processo de poda de árvores do XGBoost feito pela ferramenta.

ajustam probabilidades nó a nó, este método define uma pontuação bruta de adequação ($s_v$) baseada na topologia da árvore, onde nós mais profundos recebem pontuações maiores ($\Psi(l)$), refletindo sua menor contribuição esperada para a variância global do modelo.

A probabilidade final de corte $P(v)$ é derivada ajustando essas pontuações por um escalar global $S$, determinado via otimização numérica. Para encontrar o valor ótimo de $S$, o sistema emprega um algoritmo de Busca Binária acoplado a uma Simulação de Monte Carlo. A cada iteração da busca, o algoritmo projeta um vetor de probabilidades candidato e executa simulações rápidas com um conjunto de sondas aleatórias (vetores de indicies de *threads*). Essas sondas estimam o número total de nós que seriam removidos na prática, contabilizando corretamente os cortes por consequência (onde a remoção de um nó pai elimina implicitamente toda a subárvore descendente).

O objetivo é encontrar o $S$ que minimiza a divergência entre a taxa de corte simulada e a meta global ($R_{target}$), mantendo as probabilidades distribuídas em uma faixa contínua ("zona cinzenta"). A probabilidade final é dada pela Equação 3.1:

$$P(v) = \min\left(0.99, \max\left(0, S \cdot \Psi(l)\right)\right) \tag{3.1}$$

Na função, $\Psi(l)$ define o perfil de sensibilidade à profundidade. A ferramenta implementa estratégias distintas para esta função. Na estratégia **Linear**, a pontuação cresce aritmeticamente com a profundidade ($\Psi_{lin} = \beta + l\alpha$). Na estratégia **Exponencial**, a pon-

tuação aumenta geometricamente ($\Psi_{exp} = \beta(1+\alpha)^l$), concentrando a poda agressivamente nos nós terminais. Os valores $\alpha$ e $\beta$ são parâmetros ajustáveis que controlam o *bias* e a importância de cada nível, respectivamente. A estratégia **Flat** atua como uma máscara de zona, zerando a pontuação para nós superficiais e aplicando um valor constante para nós profundos, enquanto a calibração do escalar $S$ garante que a meta global seja atingida independentemente do perfil escolhido.

A Tabela 6 resume as características das funções de pontuação implementadas.

Tabela 6 – Estratégias de Pontuação Topológica ($\Psi$) para Otimização Global

| **Estratégia** | **Perfil de Crescimento** | **Foco da Poda** | **Dependência de $S$** |
|---|---|---|---|
| Linear | Aritmético ($\propto l$) | Distribuído | Linear |
| Exponencial | Geométrico ($\propto c^l$) | Nós Profundos | Linear |
| Flat | Degrau (Binário) | Zona Inferior | Escalar Puro |

*Nota: Todas as estratégias passam pela calibração global por Monte Carlo para determinar o escalar S.*

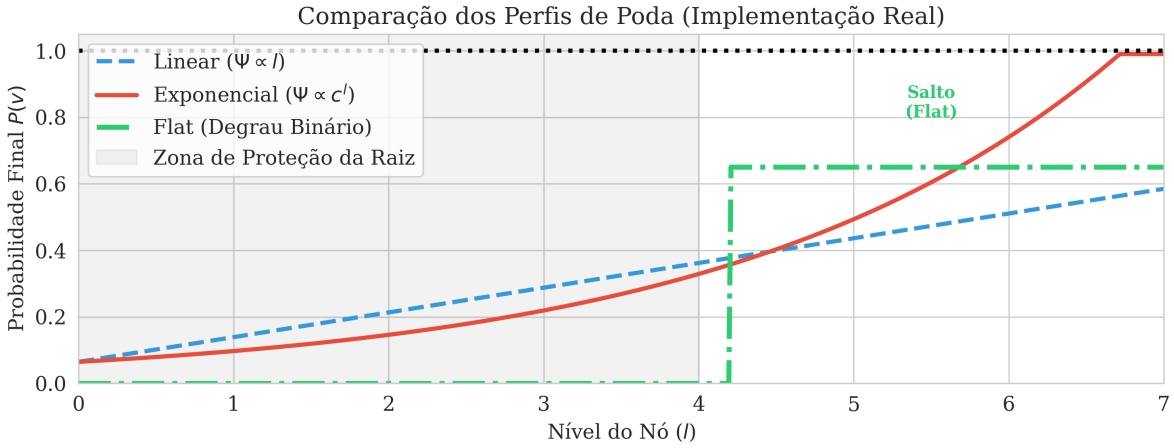



Figura 11 – Comparação da probabilidade de poda $P(v)$ para as diferentes estratégias.

A Figura 11 demonstra o comportamento da probabilidade final $P(v)$ em relação à profundidade que o nó $v$ está na árvore, sendo a profundidade máxima igual a 7. No exemplo, o escalar $S$ é 0,65, e os valores $\alpha = 0.8$ e $\beta = 0.1$. A estratégia linear está em azul, com um crescimento da probabilidade constante; a estratégia exponencial está em vermelho, mostrando que a probabilidade de corte chega ao máximo (0.99) nos nós mais profundos, mas ainda tendo uma chance considerável de poda nos níveis mais rasos; e a estratégia Flat está em verde, mostrando um salto abrupto quanto o nível $l$ do nó chega em um limiar de 60%, fazendo com que a probabilidade assuma o valor do escalar puro. A zona em cinza identifica a área rasa da árvore, próxima à raiz, onde um corte pode provocar um abatimento de inúmeros nós por consequência.

Portanto, o vetor de probabilidades $P$ tem tamanho $1 \times N_{nodes}$, onde $N_{nodes}$ representa o número total de nós na floresta. Cada *thread* da GPU utiliza esse vetor para

decidir, de forma independente, quais nós podar. Para garantir que cada *thread* tenha uma configuração única de corte, é utilizado uma função de *hash* determinística baseada no algoritmo *MurmurHash3* (JARZYNSKI; OLANO, 2020). Esta função recebe como entrada uma semente composta pelo identificador do nó e o índice da *thread*, aplicando sobre eles uma sequência de operações de deslocamento lógico (*XOR-shift*) e multiplicações por constantes, retornando um valor pseudoaleatório uniformemente distribuído entre 0 e 1. Se esse valor for menor que a probabilidade $P(v)$, o nó $v$ é selecionado para poda.

Tabela 7 – Exemplificação da Decisão de Poda Determinística via Hash.

| **Nó** ($v$) | **Probabilidade** Fixa ($P(v)$) | **Thread** ($t$) | **Saída do Hash** ($h \in [0,1]$) | **Condição** ($h < P(v)$?) | **Decisão** Final |
|---|---|---|---|---|---|
| **101** | 0.80 | 1 | 0.45 | Sim | **Podar** |
| | | 2 | 0.92 | Não | Manter |
| | | 3 | 0.12 | Sim | **Podar** |
| **42** | 0.15 | 1 | 0.10 | Sim | **Podar** |
| | | 2 | 0.67 | Não | Manter |
| | | 3 | 0.35 | Não | Manter |

A Tabela 7 exemplifica operacionalmente este processo. Para o nó $v = 101$, que possui uma alta probabilidade de poda ($P = 0.80$), a Thread 1 gera um hash $h = 0.45$ e executa o corte, enquanto a Thread 2, devido ao seu identificador distinto, gera um hash $h = 0.92$ e preserva o nó. Isso demonstra a independência estocástica garantida pela função de hash, mesmo partindo de um vetor de probabilidades estático.

## 3.2 RDSF

Esta seção descreve as melhorias implementadas na ferramenta *RDSF: Random Decision Single Forest* (BARROS et al., 2023), responsável pelo mapeamento de modelos de *Random Forest* em FPGAs utilizando *Binary Decision Diagrams* (BDDs). A versão original da ferramenta gerava equações com um número excessivo de termos, o que onerava o compilador durante o processo de construção do BDD.

Em síntese, a transformação de um modelo de árvore em funções Booleanas ocorre em três camadas. A primeira camada consiste nos comparadores entre o atributo de entrada e o limiar de cada nó. Considere, por exemplo, o nó $n_1$ com o atributo e a condição de limiar $x > 1$. Esta inequação deve ser convertida em uma função Booleana. Para fins ilustrativos, assume-se uma largura de dados de apenas 2 bits.

Para um limiar $L = 1$ (representado binariamente como $01_2$), as entradas $x_1x_0$ resultam na Tabela 8.

Ao ser processada por um **BDD (Binary Decision Diagram)**, esta equação é simplificada. Visto que ambos os ramos de $x_0$ conduzem ao terminal verdadeiro quando

Tabela 8 – Tabela verdade para a condição $x > 1$.

| $x_1$ | $x_0$ | **Valor Decimal ($x$)** | $x > 1$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 2 | 1 |
| 1 | 1 | 3 | 1 |

$x_1 = 1$, a variável $x_0$ torna-se redundante e é eliminada, resultando em:

$$c = x_1 \tag{3.2}$$

É possível reescrever genericamente o operador $>$ para derivar funções Booleanas a partir de qualquer limiar $L$ e entrada $X$. A expressão para $X > Limiar$ é dada por:

$$f(X, L) = x_1\bar{l_1} \vee (\bar{x_1}x_0\bar{l_1}\bar{l_0}) \vee (x_1x_0\bar{l_0}) \tag{3.3}$$

O primeiro termo, $x_1\bar{l_1}$, reflete a condição de dominância do bit mais significativo (MSB). Caso o MSB de $x$ seja 1 e o do limiar seja 0, a expressão é verdadeira independentemente dos bits menos significativos. O segundo termo, $\bar{x_1}x_0\bar{l_1}\bar{l_0}$, representa o cenário onde o limiar superior é 0, exigindo que o bit inferior de $x$ supere o limiar correspondente para satisfazer a condição "maior que". O último termo cobre a situação de empate no MSB, onde a decisão depende do LSB ser maior que o bit correspondente do limiar.

De forma análoga, constroem-se os operadores para a soma e a função de maioria, além das equações estruturais das árvores.

A segunda fase do fluxo de processamento envolve a utilização de BDDs para implementar a lógica da função de maioria. Neste estágio, emprega-se uma codificação do tipo *one-hot* com $n$ elementos — onde $n$ corresponde ao número de classes, para assim determinar a classe vencedora.

Para cada classe $i$, constrói-se um BDD, denominado $mv_i$, que representa a função Booleana responsável por avaliar se o somatório dos votos atribuídos à classe $i$ é superior ou igual ao somatório de cada uma das demais classes presentes na base de dados. A generalização desta etapa baseia-se nas definições apresentadas por Ferreira (FERREIRA, 1994), que introduz funções recursivas para os operadores de comparação ($>$) e igualdade ($=$) aplicados a números de $k$ bits. As relações lógicas fundamentais são descritas a seguir:

$$A_{1:1} > B_{1:1} = A_1 \wedge B_1 \tag{3.4}$$

$$A_{1:k} > B_{1:k} = (A_k \wedge B_k) \vee ((A_{1:k} > B_{1:k}) \wedge (A_k \vee \bar{B}_k)), \quad \forall k = 2, \ldots, m \tag{3.5}$$

$$A_{1:k} = B_{1:k} = \overline{(a_0 \oplus b_0)} \wedge \overline{(a_1 \oplus b_1)} \wedge \cdots \wedge \overline{(a_k \oplus b_k)} \tag{3.6}$$

É importante ressaltar que, devido à natureza da comparação paralela, a codificação resultante nos BDDs de saída não garante a exclusividade de ativação (propriedade estrita de *one-hot*), podendo resultar em mais de um bit ativado simultaneamente em casos de empate na votação. Nessas situações, a resolução do vencedor é delegada à fase subsequente de redução. Notavelmente, as variáveis que compõem esses BDDs de votação são dependentes exclusivamente das variáveis primárias do sistema (atributos de entrada), encapsulando implicitamente toda a lógica das árvores de decisão e da contagem de votos.

A fase final do processo consiste em um codificador de prioridade (*priority encoder*) que recebe o vetor de bits proveniente dos BDDs de votação e determina a classe final. A implementação desta etapa adota a estratégia de generalização proposta em (ABDELHADI; LEMIEUX, 2015), estruturando o codificador de forma recursiva e hierárquica.

No método proposto, a entrada de largura $n$ é segmentada em fragmentos de tamanho $k = 4$. Cada fragmento é processado por módulos de prioridade menores, cujas saídas são então direcionadas a um multiplexador. Paralelamente, bits de validação são gerados e encaminhados para um codificador de prioridade de nível superior, cuja saída atua tanto como sinal de seleção para o multiplexador quanto como os bits mais significativos (MSBs) do índice da classe vencedora.

Nesta implementação, o critério de desempate atribui prioridade à classe com o menor índice numérico. Ao final desta etapa, obtém-se um BDD único e consolidado que representa integralmente o modelo de classificação: dada uma amostra de entrada, a estrutura retorna diretamente o índice binário da classe predita.

As melhorias implementadas na ferramenta RDSF focaram na otimização do processo de geração das equações Booleanas, reduzindo o número de termos e, consequentemente, a complexidade do BDD resultante. Isso foi alcançado aplicando fatoração algébrica e eliminando redundâncias durante a construção das expressões. Como resultado, o compilador enfrenta uma carga reduzida, diminuindo o tempo de síntese e o consumo de recursos no FPGA.

## 3.3 TreeLUT

Nesta seção será apresentada modificações e extensões da ferramenta *TreeLUT* (KHATAEI; BAZARGAN, 2025), que mapeia modelos de *XGBoost* em *FPGAs* aplicando

estratégias de quantização no modelo já treinado. O objetivo da ferramenta é utilizar essas estratégias para reduzir a complexidade computacional e a área de hardware usada durante a inferência do modelo, sem comprometer significativamente a qualidade preditiva.

A quantização é aplicada em duas etapas principais: a quantização dos dados de entrada e a quantização das pontuações de cada folha das árvores. Ambas as quantizações são parametrizadas pelo número de bits que representarão os novos valores. A quantização dos dados de entrada envolve a redução da precisão dos dados separados por atributos, representando-os com um número menor de bits ($w_{feature}$). Isso é feito através da normalização dos dados e mapeamento para um intervalo discreto, usando quantização *Min-Max*(MISHCHENKO et al., 2019)q.

A Figura 12 ilustra o processo de quantização utilizando *Min-Max* para $w_{feature} = 2$. Um valor inteiro de entrada $DATA$ de 32 bits é propagado para uma cascata de comparadores $(Q1, Q2, Q3)$ no qual cada limiar é definido como:

$$LIM_i = MIN + \left( \frac{(MAX - MIN)}{2^{w_{feature}}} \right) \cdot i \tag{3.7}$$

onde $MIN$ e $MAX$ são os valores mínimo e máximo observados no atributo durante o treinamento do modelo, e $i$ varia de 1 a $2^{w_{feature}} - 1$. O valor de entrada é comparado sequencialmente com esses limiares, e o primeiro comparador que for satisfeito determina o intervalo no qual o valor original se encontra.

O resultado da quantização é um valor discreto $Q\_DATA$ de 2 bits, que representa o intervalo no qual o valor original se encontra. Esse processo reduz a quantidade de bits necessários para representar cada atributo, diminuindo a complexidade dos comparadores das árvores em hardware, mas pode reduzir a acurácia do modelo. Pois o modelo só é treinado, depois do ajuste da quantização.

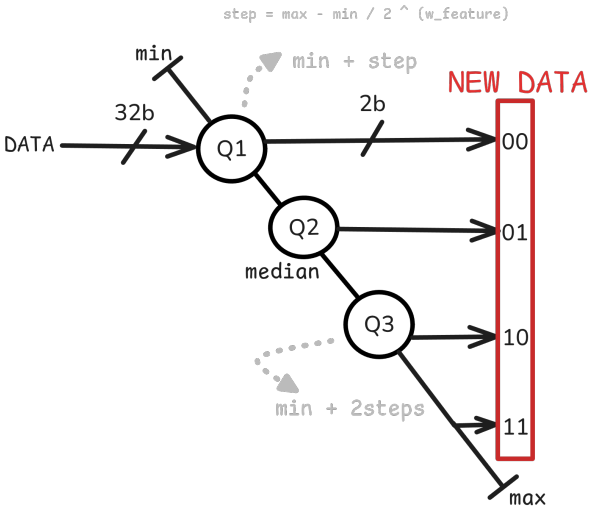



Figura 12 – Processo de quantização Min-Max.

A segunda estratégia de quantização é aplicada nas pontuações das folhas das árvores, após o treinamento do modelo. O mesmo processo de quantização é utilizado, mas agora o número de bits é definido por $w_{tree}$ e o espaço de valores é baseado na pontuação mínima e máxima observada nas folhas da floresta de cada classe $k$. A redução de precisão, neste caso, é justificada pelo fato de que, em problemas de classificação, a decisão final é baseada na comparação relativa entre as pontuações das classes, e não nos valores absolutos. Portanto, uma representação com menor precisão pode ser suficiente para manter a qualidade preditiva do modelo.

A partir das quantizações pré-definidas, a ferramenta gera as descrições de *hardware* em *Verilog* e *VHDL* (nova funcionalidade proposta). O processo abrange a criação dos módulos de comparadores, quantização, árvores, somadores e *argmax*, além da estrutura de interconexão e verificação. Visando corrigir o erro gerado pela quantização dos pesos, utiliza-se um *bias* nos acumuladores de pontuação das florestas. Esse termo de correção é obtido pela soma das diferenças entre o valor original e o valor quantizado de todas as folhas pertencentes àquela floresta. A Tabela 9 exemplifica o processo de quantização e a formação do termo de *bias* para uma configuração com $W_{tree} = 3$ em uma floresta de duas árvores. O procedimento é detalhado em três etapas sequenciais: o Passo 1 realiza o deslocamento dos valores, subtraindo o menor valor das folhas e acumulando-o no *bias* global; o Passo 2 efetua o escalonamento, mapeando o maior valor contínuo para o limite máximo representável com 3 bits ($111_2$ ou 7); e o Passo 3 conclui com a discretização, arredondando os valores resultantes para o inteiro mais próximo.

Tabela 9 – Exemplo numérico da aplicação da quantização das folhas e geração do valor *bias*, retirado de (KHATAEI; BAZARGAN, 2025).

| **Passo** | **Bias** | **Árvore de Decisão 1** | | | | **Árvore de Decisão 2** | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Value | $v_0$ | $v_1$ | $v_2$ | $v_3$ | $v_0$ | $v_1$ | $v_2$ | $v_3$ |
| Original | 0.00 | 2.00 | -0.10 | 0.50 | -0.70 | -0.40 | 0.80 | -1.40 | 0.00 |
| Passo 1 | -2.10 | 2.70 | 0.60 | 1.20 | 0.00 | 1.00 | 2.20 | 0.00 | 1.40 |
| Passo 2 | -5.44 | 7.00 | 1.56 | 3.11 | 0.00 | 2.59 | 5.70 | 0.00 | 3.63 |
| Passo 3 | -5 | 7 | 2 | 3 | 0 | 3 | 6 | 0 | 4 |

A primeira diferença entre a implementação original(KHATAEI; BAZARGAN, 2025) e implementação proposta neste trabalho se dá pela forma com que cada árvore é representada em hardware. Na ferramenta original, as árvores são implementadas utilizando lógica de equações Booleanas para representar os caminhos de decisão. Entretanto, o gerador produzia equações excessivamente extensas, as quais dificultavam a análise sintática do código Verilog resultante, além de empregar construções que oneravam o processo de síntese. A primeira modificação consistiu na decomposição das equações em funções menores e na inclusão da opção de geração baseada em multiplexadores. Tais melhorias resultaram na redução do tempo de síntese e da área ocupada no *FPGA*. Essa diferença

de abordagem implica em uma redução da área a ver o processo de síntese e também o tipo de hardware hospedeiros(SILVA et al., 2024).

A segunda modificação é quanto ao que é considerado como parte do modelo mapeado, isto é, quais módulos são contabilizados na área total do modelo. Na implementação original, apenas as árvores, comparadores e somadores são considerados, tratando a quantização das entradas e o cálculo do *argmax* como componentes externos ou inerentes ao sistema. Embora recorrente na literatura (ANDRONIC; CONSTANTINIDES, 2025a; BACELLAR et al., 2024a), essa abordagem tende a subestimar a complexidade real do circuito ao abstrair custos significativos de interface e pós-processamento. Na implementação proposta, todos os módulos necessários para a inferência do modelo são contabilizados, incluindo a quantização das entradas e o cálculo do *argmax* para fornecer uma estimativa mais realista da ocupação de *hardware*, permitindo uma avaliação precisa do *trade-off* entre precisão e eficiência, além de evidenciar novas oportunidades de otimização sistêmica.

A Figura 13 ilustra o fluxo completo de processamento da ferramenta, abrangendo desde a etapa de quantização dos dados até a geração do código final para validação. O diagrama utiliza um código de cores para diferenciar as contribuições deste trabalho em relação à ferramenta original (KHATAEI; BAZARGAN, 2025). Os módulos em vermelho representam os componentes adicionados à métrica de contabilização de LUTs: a quantização de entrada e o cálculo do *argmax*, módulos (A) e (E). Os blocos em azul indicam as etapas modificadas ou estendidas nesta proposta, especificamente as estratégias de implementação das árvores via multiplexadores e equações, modulo (C). Por fim, os módulos em cinza correspondem à infraestrutura preservada da implementação original, módulos (B) e (D).

Neste exemplo, o modelo de *XGBoost* visa classificar um *Dataset D* com quatro atributos e três classes objetivo utilizando três árvores com profundidade máxima também três, sendo assim, há quatro módulos de quantização, um módulo de comparação, três módulos de soma, um para cada classe e um módulo de *argmax*. Os parâmetros de quantização são $W_{feature} = 2$ e $W_{tree} = 3$.

## 3.4 Go-Fast

Nesta seção, será apresentada a ferramenta *Go-Fast – GPU Optimizer for FPGA-LUT Approximate Simulation and Tuning*(BARROS et al., 2025), elaborada durante o desenvolvimento deste trabalho. A ferramenta tem como objetivo otimizar a simulação aproximada de circuitos digitais baseados em LUTs em *FPGAs*, utilizando *GPUs* para acelerar o processo de simulação, além de aplicar estratégias de poda para reduzir a área ocupada no *FPGA*. A seguir, são detalhados os principais componentes e funcionalidades

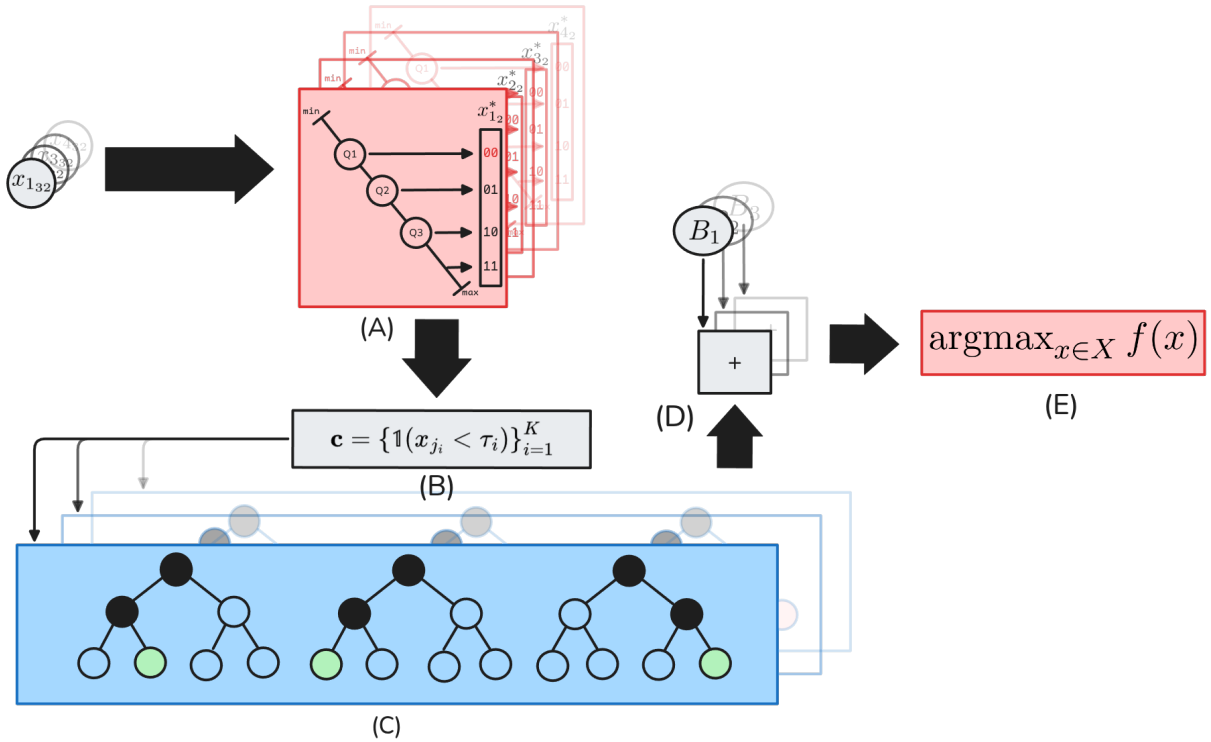



Figura 13 – FLuxo da ferramenta TreeLUT.

da ferramenta.

Pensando em uma ferramenta de síntese lógica aproximada, a simulação rápida, visando verificação e validação de qualidade, é fundamental. É neste contexto, onde o ciclo "otimização, simulação, avaliação"é refeito múltiplas vezes, que a ferramenta Go-Fast se insere. A Figura 14 apresenta a arquitetura geral da ferramenta, composta por quatro módulos principais. O primeiro, responsável pelo mapeamento, recebe o circuito descrito em *Verilog HDL* e, utilizando as ferramentas externas Yosys/ABC, faz a conversão em um grafo de LUTs que representa a estrutura lógica do hardware. O segundo módulo é o *Graph Parser*, que interpreta esse grafo, extraindo informações topológicas e de conectividade importantes para a simulação sequencial em GPU.

Em seguida, o Gerador de Código cria o código CUDA. Adicionalmente, este módulo atua como uma ferramenta de simulação a nível de LUT e seu desempenho pode ser comparado com os tempos de execução de outros simuladores de Verilog como *Verilator*. Por fim, o módulo de Poda aplica diferentes variantes de corte ao grafo de LUTs, uma por *Thread*, e gerência a execução das simulações, avaliando o impacto de cada variante na qualidade final do circuito. Atualmente, essa abordagem explora a força bruta da GPU para navegar pelo espaço de busca; no entanto, dada a magnitude massiva desse espaço, o processo de busca poderia ser refinado no futuro com estratégias mais inteligentes.

Os estímulos de entrada podem ser gerados de três formas distintas: (i) aleatoriamente, onde cada amostra é independente e distribuída uniformemente utilizando a função *curand*(NVIDIA; VINGELMANN; FITZEK, 2020) disponível no framework pa-

drão de trabalho do CUDA; (ii) sequencialmente, onde as amostras são geradas em uma sequência incremental; e (iii) a partir de um conjunto de dados fornecido pelo usuário, permitindo a simulação baseada em cenários reais ou específicos, o que é o mais adequado para uma avaliação de um mapeamento de um modelo de aprendizado de máquina. Essa flexibilidade na geração de estímulos permite que a ferramenta seja adaptada a diferentes necessidades de simulação e validação.

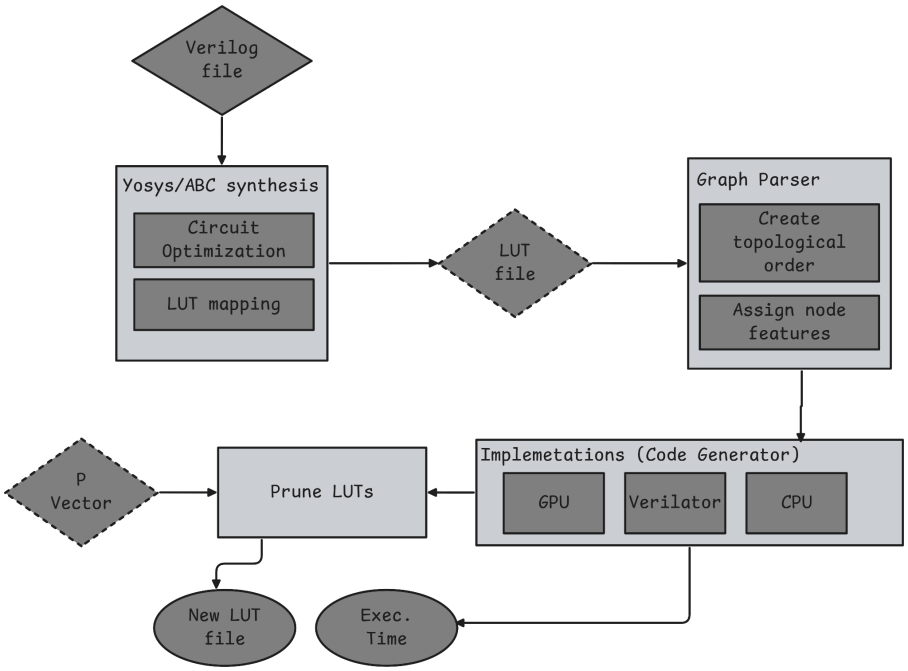



Figura 14 – Arquitetura da ferramenta Go-Fast.

O fluxo de operação do *kernel* de simulação é ilustrado na Figura 15. Inicialmente, os dados de entrada são organizados para explorar o paralelismo em nível de bits(AGOSTA; BARENGHI; PELOSI, 2012). Seja $M$ o número de entradas primárias do circuito, o conjunto de estímulos $S$ é definido como:

$$S = \{V_1, V_2, \ldots, V_M\} \tag{3.8}$$

onde cada elemento $V_j \in \{0,1\}^{32}$ representa o vetor de estados da $j$-ésima entrada do circuito. Desta forma, cada bit $n$ (com $0 \leq n < 32$) do vetor $V_j$ corresponde ao valor lógico daquela entrada para a $n$-ésima amostra de simulação independente. A título de exemplo, considere-se um circuito elementar composto por cinco entradas $V_0, V_1, \ldots, V_4$. Para a simulação, é possível gerar todas as combinações de valores $(0, 1, \ldots, 31)$ ou restringir-se a um subconjunto representativo.

Visando a otimização computacional, emprega-se a técnica de empacotamento de bits em palavras de 32 bits. No exemplo supracitado de cinco variáveis, sob uma abordagem de simulação tradicional, cada linha da tabela verdade corresponderia a uma combinação única de estados lógicos (variando de 0 a 31).

Tabela 10 – Enumeração dos estados para 5 variáveis.

| **Estado** | $v_4$ | $v_3$ | $v_2$ | $v_1$ | $v_0$ |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 1 |
| 2 | 0 | 0 | 0 | 1 | 0 |
| 3 | 0 | 0 | 0 | 1 | 1 |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |
| 31 | 1 | 1 | 1 | 1 | 1 |

Mediante a transposição para a representação vertical, cada variável $V_i$ é mapeada em um único inteiro de 32 bits. Nessa estrutura, o bit na posição $n$ do inteiro representa o valor lógico da variável $V_i$ na Tabela 11 correspondente ao estado $n$ da tabela original 10.

Tabela 11 – Transposição de variáveis para vetores de bits (formato *Bit-Slicing*).

| **Variável** | **Vetor de Bits (Binário de 32-bits)** |
|---|---|
| $V_0$ | $01010101010101010101010101010101_2$ |
| $V_1$ | $00110011001100110011001100110011_2$ |
| $V_2$ | $00001111000011110000111100001111_2$ |
| $V_3$ | $00000000111111110000000011111111_2$ |
| $V_4$ | $00000000000000001111111111111111_2$ |

A transformação para o formato transposto (*Bit-Slicing*) permite que a GPU processe múltiplos estados lógicos simultaneamente.

- **Mapeamento:** Em vez de uma matriz $32 \times 5$, utilizam-se 5 variáveis de 32 bits. O valor da variável $v_i$ no estado $j$ é extraído pelo $j$-ésimo bit do inteiro $V_i$.
- **Operação Bitwise:** Para o cálculo de uma função $f(v_0, v_1, v_2)$, aplicam-se operadores lógicos diretamente nos inteiros:

$$\text{Resultado} = (V_0 \ \& \ V_1) \mid V_2$$

Tal mecanismo viabiliza a execução da operação para todos os 32 cenários em um único ciclo de instrução. Em sistemas com maior número de variáveis, o empacotamento permite acelerar o processamento por um fator de até 32×.

Essa estruturação traz duas vantagens críticas: reduz o acréscimo de transferência de dados entre a CPU e a GPU, no contexto em que as entradas são passadas como parâmetros, e permite que as operações lógicas nas LUTs sejam executadas de forma *bitwise* sobre os vetores $V_j$, processando 32 amostras simultaneamente. Em seguida, utilizando a estratégia *More Work per Thread*, cada *thread* da GPU recebe um subconjunto de estimulos $E = \{S_1, S_2, \ldots, S_n\}$, dividindo o trabalho em número de amostras, e é responsável

por propagar esses vetores através de todo o grafo de LUTs, calculando as saídas para as 32 instâncias de simulação em um único fluxo de execução. As saídas resultantes são então armazenadas em um formato similar, facilitando a análise posterior.

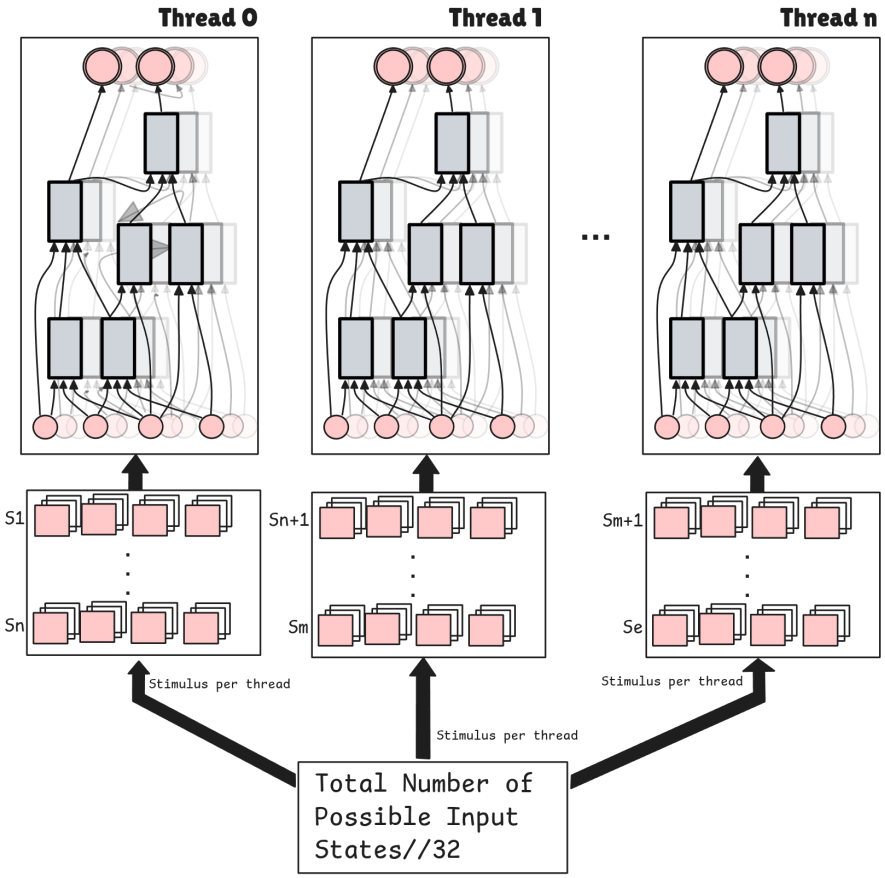



Figura 15 – Funcionamento do kernel de simulação.

Em contraste com o *kernel* de simulação, o *kernel* de poda altera a estratégia de paralelismo. Em vez de subdividir o trabalho pelo número de amostras, cada *thread* é responsável por processar todo o conjunto de estímulos possíveis para uma variante de circuito distinta. Desta forma, a divisão de trabalho ocorre por variantes de combinação de LUTs podadas.

A Figura 16 ilustra este fluxo de operação. Inicialmente, são alocados dois vetores globais binários, $P$ e $I$, ambos de dimensão $N_{LUTs} \cdot N_{Threads}$. Cada *thread* $k$ acessa um segmento destes vetores correspondente à sua variante de poda específica, definidos como:

- **Vetor de Máscara ($p_k$):** Indica quais LUTs devem ser podadas (desconectadas da lógica original) na variante $k$.
- **Vetor de Injeção ($i_k$):** Determina o valor constante (0 ou 1) que cada LUT podada passará a propagar.

Com base nessa configuração $(p_k, i_k)$, a *thread* simula o comportamento do circuito modificado para todo o conjunto de estímulos fornecido, avaliando o impacto da poda na qualidade da saída.

Quanto a construção dos vetores de poda, ela pode assumir várias estratégias (ABDELFATTAH et al., 2020), tornando-se um ponto flexível. Neste trabalho, adotou-se uma abordagem de geração aleatória, onde cada combinação de poda é criada de forma estocástica, permitindo a exploração ampla do espaço de variantes possíveis.

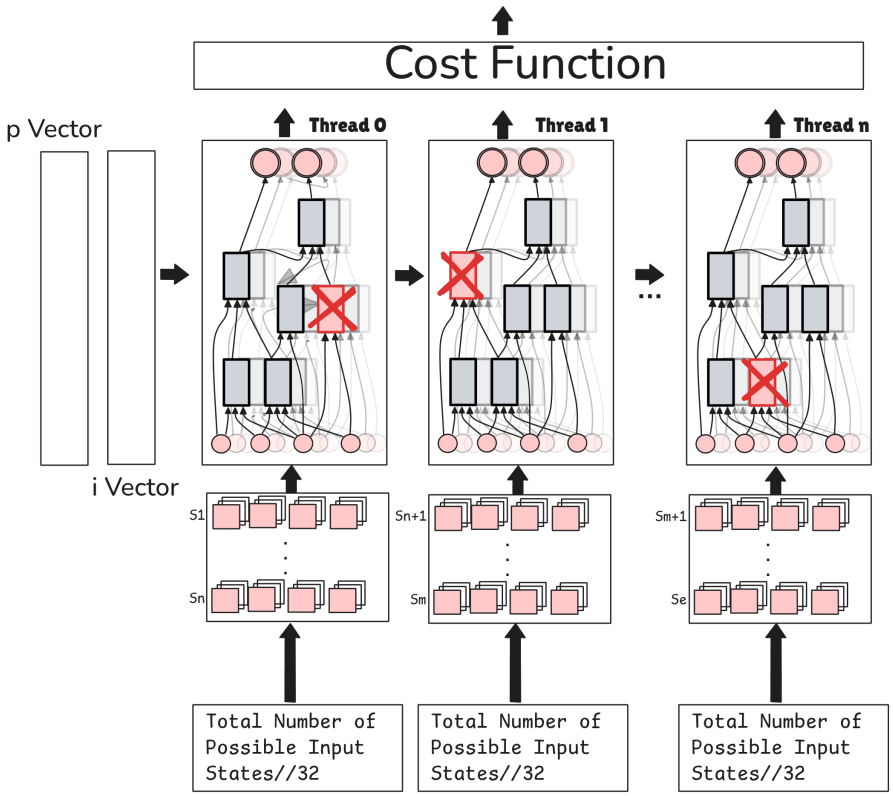



Figura 16 – Funcionamento do kernel de poda.

Uma outra distinção do *kernel* de poda em relação ao de simulação é a integração de um módulo de custo no fluxo de simulação. Este módulo avalia a qualidade do circuito podado simultaneamente à sua execução, gerando como saída um vetor de custos $C \in \mathbb{R}^T$, onde $T$ representa o número de variantes de poda.

A função de custo padrão implementada na ferramenta é a *Error Rate* (ER). Diferente de uma comparação simples de bits isolados, a métrica adotada considera a topologia de múltiplas saídas do circuito, onde uma amostra é considerada errada se houver divergência em **qualquer** uma das suas saídas. A função de custo padrão implementada é calculada através da seguinte formulação:

$$ER_k = \frac{1}{N} \sum_{j=1}^{N} \vee \left( \sum_{out=0}^{K} f(j) \oplus \sum_{out=0}^{K} \hat{f}(j) \right) \tag{3.9}$$

Nesta equação, a taxa de erro da variante $k$ é determinada verificando a consistência global das saídas para cada uma das $N$ amostras simuladas. Os somatórios internos realizam a agregação dos $K$ pinos de saída, permitindo que a comparação entre o circuito original $f(j)$ e o podado $\hat{f}(j)$ considere a palavra de saída completa. Desta forma, a função contabiliza um erro sempre que ocorre divergência em qualquer uma das saídas para um dado estímulo $j$, penalizando alterações que corrompem o comportamento funcional do vetor de saída como um todo, independentemente de quantos bits individuais foram afetados.

Essa abordagem garante que o impacto da poda seja avaliado globalmente sobre o comportamento funcional do circuito, penalizando qualquer alteração que corrompa a integridade da saída. O módulo de custo é integrado diretamente no fluxo de simulação do *kernel* de poda e, ainda, pode ser adaptado para operar em conjunto com diferentes estratégias de poda. Um modelo de classificação, por exemplo, demanda precisão na determinação da classe vencedora, tornando-o mais sensível a alterações lógicas. Em contrapartida, modelos de regressão beneficiam-se de uma tolerância natural a técnicas de computação aproximada, visto que erros introduzidos nos bits menos significativos (*LSBs*) resultam em desvios numéricos marginais que não invalidam a predição. Essa flexibilidade permite que a ferramenta seja customizada para atender a requisitos específicos de qualidade e desempenho, dependendo do contexto de aplicação do circuito podado.

# 4 Experimentos

## 4.1 Conjunto de Dados

Para a avaliação das estratégias propostas, foram selecionados quatro conjuntos de dados amplamente utilizados na literatura de aprendizado de máquina: *adult*(BECKER; KOHAVI, 1996), *covtype*(BLACKARD, 1998), *drybean*(UCI, 2020) e um conjunto sintético gerado via a função *make_classification* da biblioteca *sklearn*(PEDREGOSA et al., 2011). As características principais desses conjuntos de dados estão resumidas na Tabela 12.

Tabela 12 – Resumo das características dos conjuntos de dados utilizados.

| Dataset | Amostras | Atributos | Atributos (Num/Cat) | Classes | Fonte |
|---|---|---|---|---|---|
| Adult | 48.842 | 14 | 6 / 8 | 2 | UCI (BECKER; KOHAVI, 1996) |
| Covtype | 581.012 | 54 | 10 / 44† | 7 | UCI (BLACKARD, 1998) |
| Dry Bean | 13.611 | 16 | 16 / 0 | 7 | UCI (UCI, 2020) |
| Sintético | 1.000.000 | 30 | 20 / 0 | 4 | Scikit-learn (PEDREGOSA et al., 2011) |

†Os 44 atributos categóricos são binários, resultantes da codificação *one-hot* de variáveis de solo e área.

O conjunto de dados *adult* (também conhecido como *Census Income*) é uma referência em tarefas de classificação binária. Ele agrega dados demográficos extraídos do censo de 1994, contendo atributos mistos (categóricos e numéricos), com o objetivo de prever se a renda anual de um indivíduo excede 50 mil dólares.

O *covtype* (*Forest Cover Type*) representa um desafio de maior escala, utilizado para prever o tipo de cobertura florestal (dentre 7 classes possíveis) em áreas selvagens, baseando-se estritamente em variáveis cartográficas como elevação, declividade e tipo de solo. Com mais de 580 mil amostras, este conjunto é frequentemente utilizado para testes de estresse e escalabilidade de algoritmos.

O *drybean* consiste em características geométricas (como área, perímetro, compacidade e eixos) extraídas de imagens de alta resolução de sete variedades de feijão seco. Com 16 atributos numéricos, ele apresenta um problema de classificação multiclasse focado em morfologia, exigindo que o modelo distinga classes com variações sutis de forma e textura.

Por fim, o conjunto *sintético* foi gerado artificialmente para permitir o controle preciso das propriedades do problema, isolando variáveis de confusão. Ele serve como um cenário controlado para avaliar o comportamento do algoritmo sob condições específicas, como o equilíbrio de classes e a linearidade dos dados, sem os viéses inerentes aos dados reais.

Para facilitar o desenvolvimento e teste das ferramentas, foi elaborado uma biblioteca em *Python* para manejo dos conjuntos de dados, incluindo funções para pré-processamento, divisão em conjuntos de treino e teste, e avaliação de métricas de desempenho. Essa biblioteca está disponível no repositório GitHub do projeto (BARROS, 2025).

## 4.2 RDSF

Nesta seção, avalia-se o desempenho da ferramenta RDSF no mapeamento de modelos *Random Forest* para FPGAs utilizando BDDs. A análise concentra-se no impacto da complexidade do modelo (número de árvores e profundidade máxima) sobre o consumo de recursos lógicos (LUTs), o tamanho da representação intermediária (número de nós do BDD) e a acurácia resultante.

As Tabelas 13, 14 e 15 apresentam os resultados para os conjuntos de dados *Adult*, *Dry Bean* e *Covtype*, respectivamente. As colunas detalham a evolução da complexidade do BDD em três estágios: "Eq." (nós das árvores individuais), "Soma" (nós gerados pela operação aritmética de adição das árvores) e "Voto" (lógica de decisão final/comparação). A coluna "Final" representa o número de nós após as otimizações de minimização do BDD, e "LUTs" indica o custo final de síntese no hardware utilizando a ferramenta Yosys (WOLF, 2016).

Tabela 13 – Resultados de síntese e complexidade de nós para o conjunto de dados *Adult*.

| **Configuração da RF** | | **Número de Nós BDD** | | | | **Acurácia (%)** | **LUTs** |
|---|---|---|---|---|---|---|---|
| **Árvores** | **Prof.** | **Eq.** | **Soma** | **Voto** | **Final** | | |
| 3 | 3 | 15 | 30 | 20 | 20 | 80,79 | 13 |
| 4 | 4 | 44 | 476 | 262 | 182 | 82,94 | 155 |
| 5 | 5 | 111 | 2.413 | 1.149 | 1.149 | 83,85 | 1.046 |
| 6 | 6 | 305 | 30.527 | 16.048 | 11.989 | 85,21 | 11.509 |

Para o conjunto de dados *Adult* (Tabela 13), um problema de classificação binária, observa-se que a ferramenta escala de forma controlada até a profundidade 5. A transição da profundidade 5 para 6 resulta em um aumento de aproximadamente 10x no número de LUTs (de 1.046 para 11.509). Nota-se que a etapa de "Soma" é a que gera o maior número de nós intermediários, mas a lógica de votação binária permite uma redução eficiente na etapa "Final", com exemplo os 30.527 nós de soma para 11.989 nós finais na profundidade 6. Esse padrão se segue nos outros dois conjuntos de dados

O comportamento para problemas multiclasse revela os desafios de escalabilidade dos BDDs. No conjunto *Dry Bean* (Tabela 14), que possui 7 classes, ocorre uma explosão combinatória na profundidade 6. O número de nós de votação atinge a ordem de 1,7 mi-

Tabela 14 – Resultados de síntese e complexidade de nós para o conjunto de dados *Dry Bean*.

| **Configuração da RF** | | **Número de Nós BDD** | | | | **Acurácia (%)** | **LUTs** |
|---|---|---|---|---|---|---|---|
| **Árvores** | **Prof.** | **Eq.** | **Soma** | **Voto** | **Final** | | |
| 3 | 3 | 39 | 85 | 150 | 88 | 76,91 | 81 |
| 4 | 4 | 112 | 434 | 1.510 | 847 | 82,35 | 779 |
| 5 | 5 | 277 | 3.149 | 31.117 | 16.823 | 88,32 | 16.075 |
| 6 | 6 | 626 | 93.709 | 1.709.689 | 985.027 | 88,88 | 963.585 |

lhão, resultando em quase 1 milhão de LUTs. Isso ocorre porque a lógica de *argmax* em BDDs exige comparações pareadas entre os somatórios de todas as classes, dificultando o compartilhamento de subgrafos. Este caso ilustra o limite prático da representação pura em BDD para lógicas aritméticas complexas sem o uso de poda agressiva ou particionamento (SILVA et al., 2024).

Tabela 15 – Resultados de síntese e complexidade de nós para o conjunto de dados *Covtype*.

| **Configuração da RF** | | **Número de Nós BDD** | | | | **Acurácia (%)** | **LUTs** |
|---|---|---|---|---|---|---|---|
| **Árvores** | **Prof.** | **Eq.** | **Soma** | **Voto** | **Final** | | |
| 3 | 3 | 12 | 22 | 15 | 15 | 56,72 | 11 |
| 4 | 4 | 40 | 117 | 63 | 44 | 59,59 | 41 |
| 5 | 5 | 168 | 1.900 | 1.083 | 957 | 60,73 | 896 |
| 6 | 6 | 529 | 29.271 | 15.357 | 10.298 | 65,34 | 9.718 |

O conjunto *Covtype* (Tabela 15), apesar de também ser multiclasse, apresentou um comportamento mais moderado que o *Dry Bean*, com o custo de LUTs na profundidade 6 mantendo-se na faixa de 9.700. A diferença pode ser atribuída à estrutura das árvores e à correlação entre as variáveis, que neste caso favoreceu uma maior redução durante a otimização do BDD (de 29.271 nós de soma para 10.298 finais), já que boa parte dos atributos do conjunto de dados são binários.

## 4.3 XGB2GPU

Nesta seção, avalia-se a estratégia paralela de pós-poda em GPU. Inicialmente, a Seção 4.3.1 investiga a seleção dos hiperparâmetros $\beta$ e $\alpha$ da ferramenta, utilizando um conjunto de dados sintético com um número reduzido de árvores. Em seguida, a Seção 4.3.2 analisa as estratégias de poda, contrastando cortes diretos com podas indiretas (consequentes), além de verificar a variação de acurácia para os conjuntos de dados *adult*, *covtype*, *drybean* e sintético. O terceiro experimento, descrito na Seção 4.3.3, examina o impacto do número de *threads* na acurácia do pós-poda, considerando um conjunto

de dados sintético com maior número de árvores. Por fim, a Seção 4.3.4 sumariza os resultados obtidos, comparando métricas de tempo de execução, acurácia e *F1-Score* para cada cenário.

### 4.3.1 Escolha de Hiperparâmetros

Para a análise de sensibilidade dos hiperparâmetros $\beta$ (fator de viés/corte) e $\alpha$ (peso da importância), conduziu-se um experimento utilizando um conjunto de dados sintético gerado pela função $make\_classification$, contendo 100.000 amostras, 30 atributos e 4 classes. O modelo base, um XGBoost composto por 3 árvores com profundidade máxima de 3, serviu de referência para avaliar as estratégias de poda *Linear* e *Exponencial*, a primeira aplicando uma penalização linear baseada na profundidade e importância do nó, enquanto a segunda impõe um critério de corte exponencialmente progressivo, intensificando a remoção de ramos em níveis mais profundos (detalhadas na Seção 3.1).

A Figura 17 apresenta o mapa de calor da degradação da acurácia ($\Delta_{acc} = Acc_{original} - Acc_{podado}$), em função de $\beta$ (Eixo X) e $\alpha$ (Eixo Y). Neste contexto, valores próximos a zero (ou negativos) são ideais, indicando que a poda reduziu o modelo mantendo a fidelidade à acurácia original. Observa-se que a estratégia *linear* demonstra maior robustez, mantendo o desempenho em uma ampla faixa de configurações, com ponto ótimo em $\beta = 0.9$ e $\alpha = 0.5$. Em contrapartida, a estratégia *exponencial* revela sensibilidade acentuada, obtendo melhores resultados (preservação da acurácia) quando o peso da importância dos níveis ($\alpha$) é mantido baixo, com ponto ideal em $\beta = 0.5$ e $\alpha = 0.1$. Essa análise fundamenta a escolha de parâmetros fixos que maximizam a redução de *hardware* sem penalizar a capacidade preditiva.

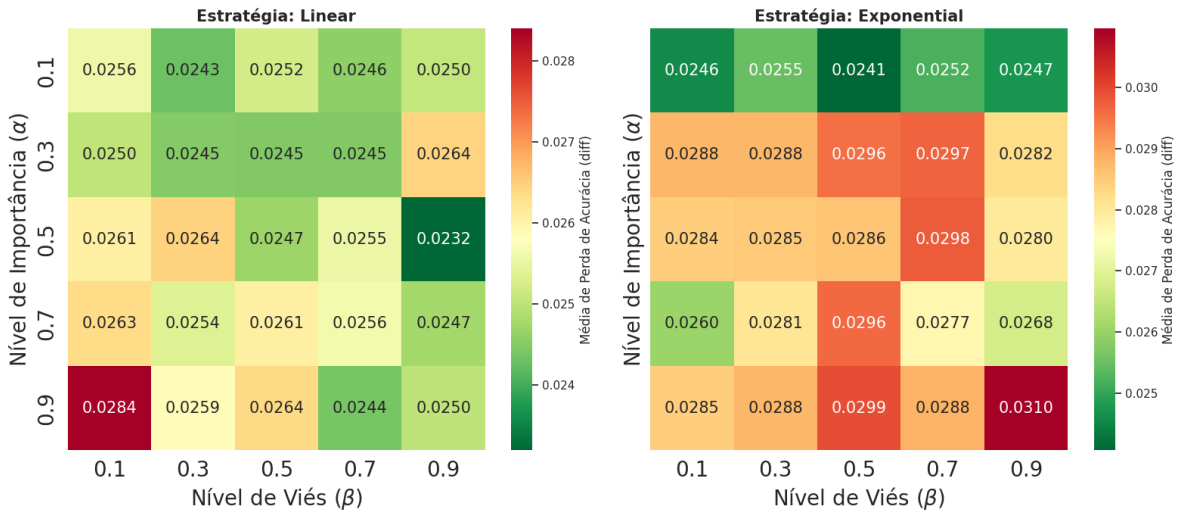



Figura 17 – Mapa de calor da diferença entre a acurácia real e a acurácia pós-poda em função dos hiperparâmetros $\beta$ e $\alpha$ para as estratégias Linear (esquerda) e Exponencial (direita).

### 4.3.2 Estratégias de Poda

Para a análise das estratégias, é fundamental distinguir a poda direta da poda consequente (ou indireta). A **poda direta** consiste na seleção de um nó para remoção, substituindo-o por uma folha cuja predição assume a média dos valores dos filhos. Em contrapartida, a **poda consequente** refere-se à eliminação em cascata dos nós descendentes: ao podar um nó pai, todos os seus descendentes são automaticamente removidos da estrutura. A Figura 18 ilustra esses mecanismos em um modelo composto por três árvores de profundidade máxima três, treinado em um conjunto de dados sintético binário. A visualização apresenta um total de seis estruturas, onde: (i) os nós em vermelho indicam os pontos de corte direto (transformados em folhas); (ii) os nós em amarelo representam os descendentes removidos por consequência; e (iii) as faixas delimitam cada uma das árvores individuais do modelo.

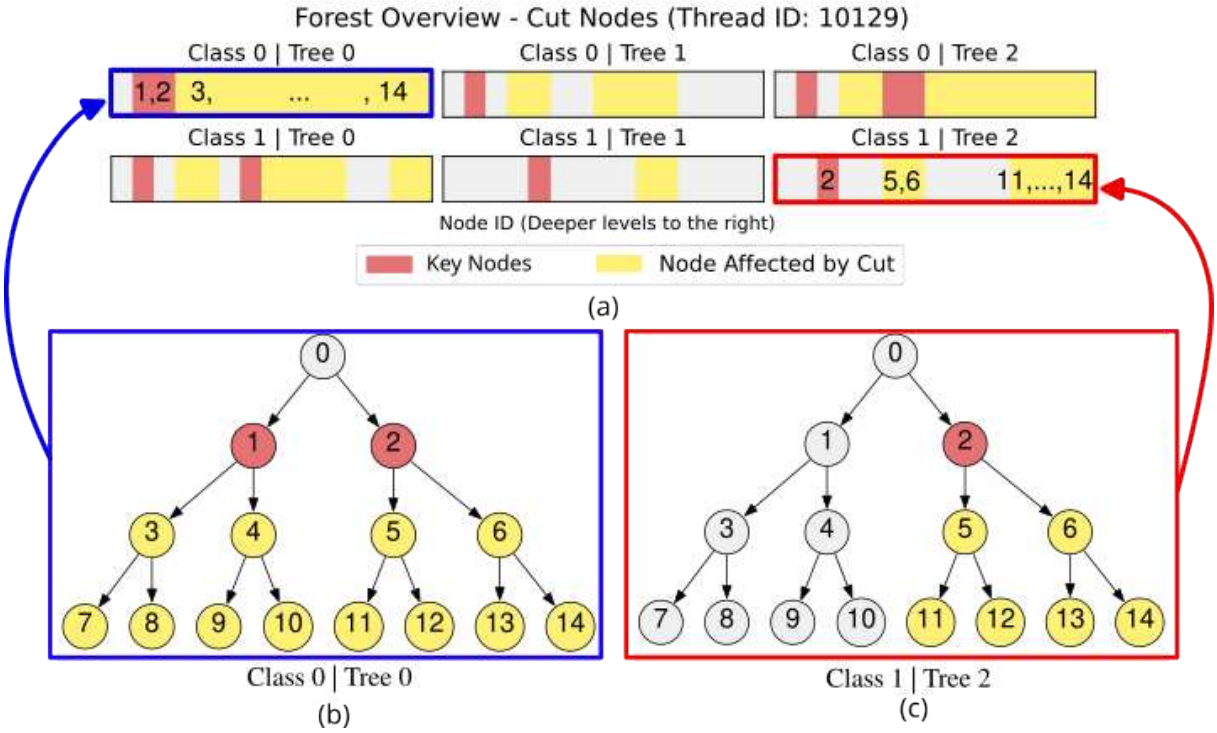



Figura 18 – Visão geral da poda de árvores.

Nesse contexto, a Figura 19 compara o impacto estrutural das três estratégias de poda implementadas no *kernel* de GPU (*Flat*, *Linear* e *Exponencial*) sobre um modelo de XGBoost com 7 árvores e profundidade máxima 7. O gráfico discrimina o percentual de nós removidos diretamente pela decisão do algoritmo (barras em vermelha) em relação ao total de nós eliminados por consequência, ou seja, a subárvore inteira descartada após o corte de um nó pai. O eixo $x$ mostra várias estratégias e porcentagem de poda, a porcentagem é referente ao número total de nós removidos, isto é, é a soma do nós diretamente removidos e os nós removidos por consequência. Observa-se que a estratégia *Flat*, ao impor uma barreira estocástica horizontal em um nível específico $l$, tende a concentrar seus cortes nas

extremidades, resultando em uma proporção maior de podas diretas. Em contrapartida, as estratégias *Linear* e *Exponencial*, que penalizam a profundidade e a importância dos nós, tendem a selecionar cortes em níveis superiores da hierarquia. Essa abordagem provoca um efeito cascata, onde a remoção de um único nó pai acarreta a eliminação de um vasto número de descendentes, justificando a predominância de nós podados por consequência nessas configurações.

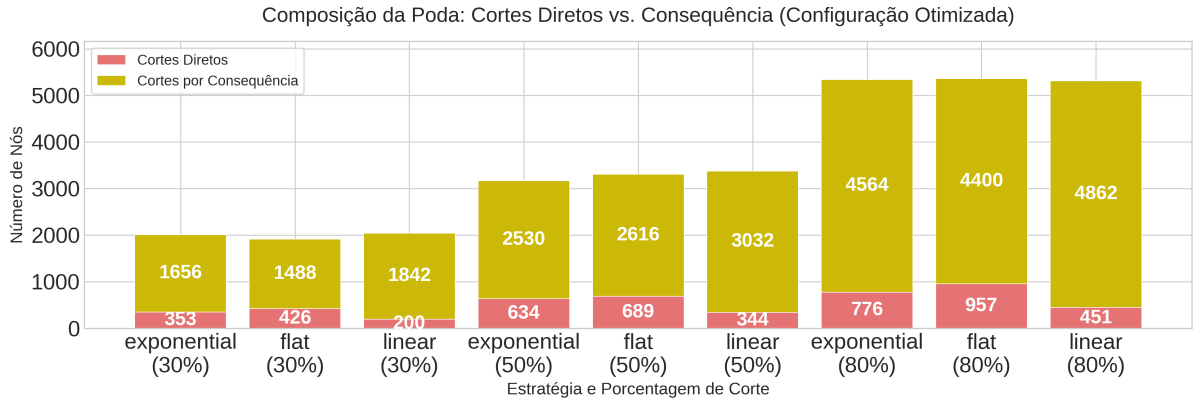



Figura 19 – Composição dos nós podados diretamente e por consequência para as estratégias Flat, Linear e Exponencial.

A Figura 20 apresenta a comparação das três estratégias de poda (Flat, Linear e Exponencial) em termos de acurácia em comparação ao número de nós podados para os conjuntos de dados *adult*, *covtype*, *drybean* e sintético, utilizando $\beta = 0.1$ e $\alpha = 0.9$. O resultado para cada uma das soluções foi retirado de uma busca exaustiva do espaço de poda, utilizando 190k threads para simular paralelamente em GPU o impacto de diferentes vetores de corte, para cada porcentagem alvo, permitindo a avaliação massiva de configurações para traçar o melhor trade-off entre acurácia e redução de nós. A linha horizontal tracejada indica a acurácia sem poda, o eixo $y$ mostra a acurácia obtida e o eixo $x$ mostra a porcentagem total de nós podados em relação ao total de nós do modelo original.

Observa-se que, em geral, a estratégia Linear tende a preservar melhor a acurácia à medida que mais nós são podados, especialmente em comparação com a estratégia Flat, invertendo os resultados em porcentagens de poda mais altos. A estratégia Exponencial também demonstra um bom desempenho, mas não tão robusto quanto a Linear. Esses resultados sugerem que considerar a importância dos níveis da árvore durante a seleção dos nós para poda é crucial para manter a capacidade preditiva do modelo, especialmente quando se busca uma redução significativa do tamanho do modelo. Um caso contraintuitivo é o conjunto de dados *adult*, onde a poda resulta em um aumento da acurácia, possivelmente devido à remoção de nós que causavam sobreajuste nas primeiras árvores.

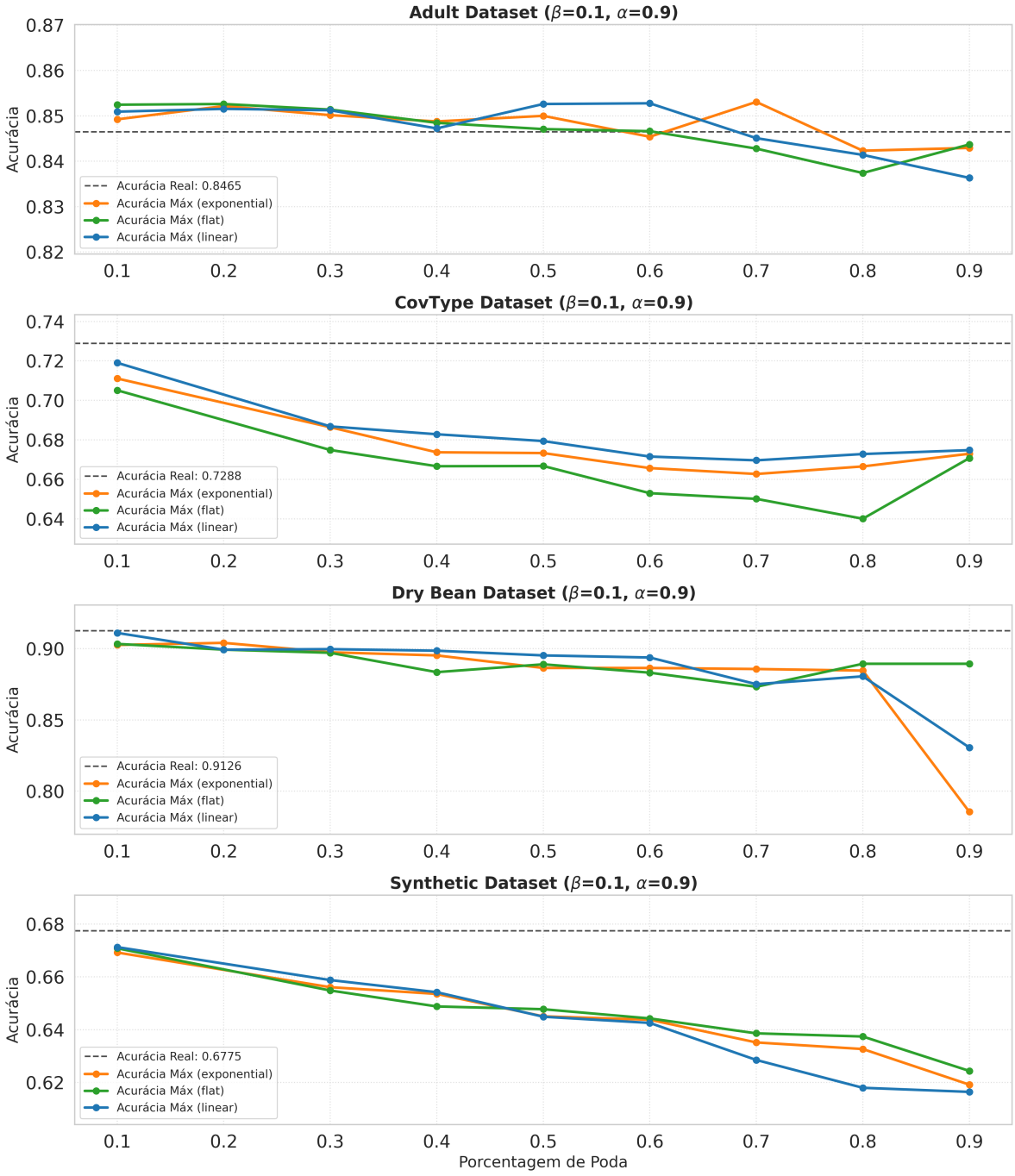



Figura 20 – Comparação das estratégias de poda (Flat, Linear e Exponencial) em termos de acurácia versus número de nós podados para os conjuntos de dados *adult*, *covtype*, *drybean* e sintético.

### 4.3.3 Impacto do número de threads na Poda

A Figura 21 ilustra a variabilidade da acurácia pós-poda em função do paralelismo empregado. No gráfico, o eixo $Y$ categoriza os experimentos pelo número total de *threads* utilizadas, o primeiro com 1.791 threads, o segundo com 19.967 e o terceiro com 194.047 threads, enquanto o eixo $x$ quantifica a acurácia obtida. As formas em "violinos" representam a distribuição de densidade das soluções encontradas: as áreas mais largas indicam onde se concentra a maioria dos resultados (neste caso, em torno da média de 51% a 52%), enquanto as extremidades afinadas representam as piores ou melhores soluções.

Para estabelecer as referências de desempenho, a linha contínua azul demarca a acurácia do modelo original sem poda (66, 21%), funcionando como o alvo ideal. Já a linha tracejada vermelha destaca a melhor solução encontrada dentre todas as simulações (64, 59%). O experimento utilizou um conjunto de dados sintético (4 classes, 10 árvores, profundidade 10) com meta de 80% de poda. A análise da distribuição revela que, embora a média de desempenho permaneça estável independentemente do número de *threads*, o aumento do volume de busca (de 1.791 para 194.047 *threads*) expande a cauda superior da distribuição. Isso resultou em 132 soluções com acurácia superior a 60% no cenário com mais *threads*, contra apenas 2 ocorrências no cenário reduzido. Esse comportamento evidencia que a densidade de soluções de alta qualidade é baixa, exigindo uma exploração exaustiva (muitas *threads*) para capturar os extremos positivos próximos à linha azul original.

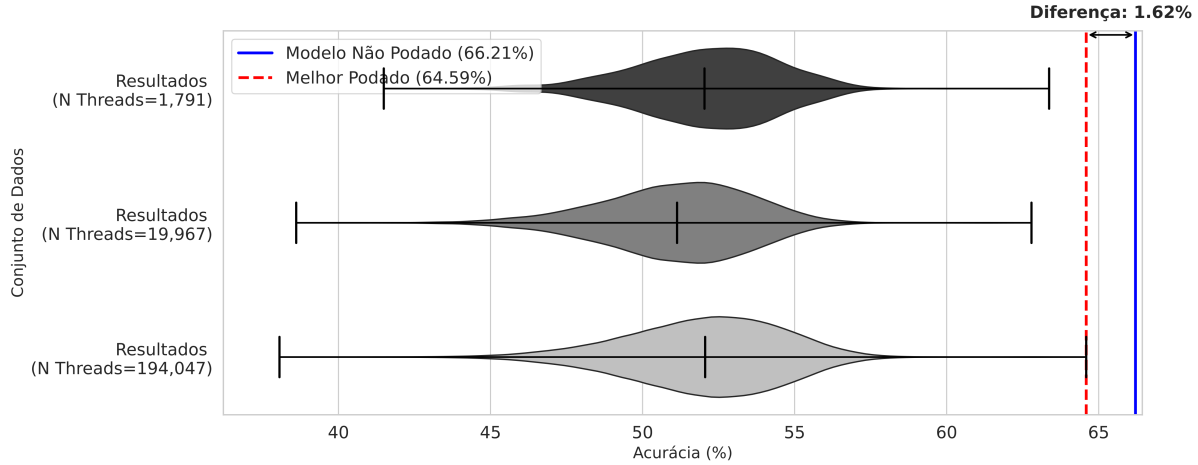



Figura 21 – Distribuição da acurácia pós-poda para diferentes números de threads.

A Tabela 16 resume as estatísticas descritivas da acurácia pós-poda para os três experimentos com diferentes números de threads. Observa-se que, embora a média da acurácia permaneça relativamente constante em torno de 51% a 52% em todas as configurações, o aumento do número de threads resulta em uma expansão significativa do intervalo entre o valor mínimo e máximo. Especificamente, o experimento com 194.047 threads alcançou um máximo de 64, 59%, contrastando com os 63, 37% do experimento

com apenas 1.791 threads. Além disso, o número de soluções com acurácia superior a 60% aumentou substancialmente, passando de apenas 2 ocorrências no menor conjunto para 132 no maior.

Esses resultados indicam que uma amostragem mais ampla através de *threads* não apenas expande a exploração do espaço de soluções, mas também eleva a probabilidade de identificar modelos podados com desempenho equivalente ao original. Tal capacidade é crucial em cenários de alta complexidade, onde o modelo apresenta uma grande quantidade de nós candidatos à poda, seja devido ao elevado número de classes, à quantidade de árvores e profundidade máxima, ou à porcentagem de poda alvo. Visto que esses fatores aumentam em escala combinatória as configurações possíveis, o número de *threads* atua como um limitante superior da amostragem; assim, mesmo que o algoritmo tenda a gerar configurações promissoras, a restrição no número de *threads* pode impedir uma exploração suficiente do espaço de busca.

Tabela 16 – Resumo Estatístico da Acurácia

| $N$ Threads | **Mín. (%)** | **Méd. (%)** | **Máx. (%)** | Número de Soluções com $\geq 60\%$ |
|---|---|---|---|---|
| 1.791 | 41,49 | 52,04 | 63,37 | 2 |
| 19.967 | 38,61 | 51,13 | 62,79 | 12 |
| 194.047 | 38,06 | 52,06 | 64,59 | 132 |

### 4.3.4 Sumário dos Resultados

Uma característica intrínseca do algoritmo XGBoost é a realização de podas nativas durante a fase de treinamento, o que impacta diretamente o número inicial de nós em cada conjunto de dados. Observa-se que o modelo para o conjunto *adult* possui 1.982 nós, o *drybean* 2.719 nós, o *covtype* 6.311 nós e o sintético 6.748 nós. Esses valores são consideravelmente inferiores ao teto teórico máximo para um modelo composto por 7 árvores com profundidade máxima 7. Aplicando a relação $N_{arv} \times (2^{D+1} - 1) \times N_{classes}$, os limites teóricos seriam de 3.570 nós para o *adult* (2 classes), 7.140 nós para o sintético (4 classes) e 12.495 nós para *drybean* e *covtype* (7 classes).

Adicionalmente, a análise da redução deve considerar a acurácia base de cada problema, definida pela probabilidade de acerto ao escolher sempre a classe majoritária. Por exemplo, o conjunto *adult* apresenta um desbalanceamento que confere uma acurácia inicial de aproximadamente 75% a um classificador ingênuo, enquanto em problemas balanceados multiclasse (como o sintético), esse patamar inicial seria de apenas 25%. Portanto, é importante ressaltar que as taxas de redução e a manutenção de acurácia apresentadas nesta seção são relativas a modelos que já passaram por essa otimização prévia do treinamento e devem ser interpretadas à luz da complexidade intrínseca de cada conjunto de dados.

A Tabela 17 detalha a avaliação para o conjunto de dados *adult* com a estratégia Exponencial. Observa-se um ganho de velocidade expressivo, superior a 3000x em relação à implementação em CPU. Nota-se que, mesmo sob uma redução agressiva de nós (aproximadamente 89,5%), a acurácia se mantém estável, oscilando minimamente em torno da base de 84,65%, o que demonstra a robustez da estratégia Exponencial em identificar nós pouco representativos.

Tabela 17 – Avaliação de poda para o conjunto de dados Adult usando 7 árvores com profundidade máxima 7 (Estratégia Exponencial). Total de Nós: 1982, Acurácia Base: 84,65%.

| **Tempo (s)** | **Speedup** | **Acurácia (%)** | **F1 (%)** | **Chave** | **Nós Removidos** |
|---|---|---|---|---|---|
| 0,1159 | 2966,91x | 84,92 | 84,47 | 25 | 169 (8,5%) |
| 0,1123 | 3062,56x | 85,01 | 84,42 | 96 | 602 (30,4%) |
| 0,1123 | 3061,47x | 85,00 | 84,00 | 164 | 884 (44,6%) |
| 0,1123 | 3061,20x | 85,31 | 84,79 | 206 | 1358 (68,5%) |
| 0,1124 | 3059,56x | 84,29 | 83,35 | 120 | 1774 (89,5%) |

Para o conjunto de dados *Dry Bean*, avaliado na Tabela 18 com a estratégia *Flat*, os resultados indicam uma *trade-off* um pouco mais acentuada. Embora o ganho de velocidade se mantenha na ordem de 1600x, a acurácia sofreu uma degradação progressiva, caindo de 91,26% (base) para 87,33% quando cerca de 69% dos nós foram removidos.

Tabela 18 – Avaliação de poda para o conjunto de dados Dry Bean usando 7 árvores com profundidade máxima 7 (Estratégia Flat). Total de Nós: 2719, Acurácia Base: 91,26%.

| **Tempo (s)** | **Speedup** | **Acurácia (%)** | **F1 (%)** | **Chave** | **Nós Removidos** |
|---|---|---|---|---|---|
| 0,0873 | 1646,09x | 90,34 | 90,34 | 48 | 192 (7,1%) |
| 0,0873 | 1646,09x | 89,72 | 89,69 | 191 | 857 (31,5%) |
| 0,0873 | 1646,47x | 88,91 | 88,92 | 287 | 1345 (49,5%) |
| 0,0872 | 1647,79x | 87,33 | 87,39 | 251 | 1885 (69,3%) |
| 0,0872 | 1648,55x | 88,95 | 88,93 | 263 | 1879 (69,1%) |

Os resultados para o conjunto de dados *Covtype*, o maior entre os avaliados com dados reais (6311 nós), são apresentados na Tabela 19 utilizando a estratégia Exponencial. Este cenário mostrou-se mais desafiador: uma poda de 41,2% dos nós resultou em uma queda de acurácia de aproximadamente 5,5 pontos percentuais (de 72,88% para 67,37%). O ganho de velocidade obtido foi próximo a 975x, um valor menor comparado aos conjunto de dados anteriores, possivelmente devido à maior complexidade da estrutura das árvores e a sobrecarga de transferência de dados na GPU para modelos maiores.

Por fim, a Tabela 20 exibe os dados para o conjunto Sintético com a estratégia *Flat*. Com um ganho de velocidade consistente em torno de 1000x, observou-se um comportamento linear na degradação da acurácia conforme o aumento da taxa de poda. A

Tabela 19 – Avaliação de poda para o conjunto de dados Covtype usando 7 árvores com profundidade máxima 7 (Estratégia Exponencial). Total de Nós: 6311, Acurácia Base: 72,88%.

| **Tempo (s)** | **Speedup** | **Acurácia (%)** | **F1 (%)** | **Chave** | **Nós Removidos** |
|---|---|---|---|---|---|
| 7,76 | 974,17x | 71,11 | 70,44 | 92 | 412 (6,5%) |
| 7,76 | 974,28x | 67,37 | 66,32 | 457 | 2597 (41,2%) |
| 7,71 | 980,05x | 67,33 | 66,05 | 520 | 3036 (48,1%) |
| 7,73 | 977,12x | 66,27 | 66,45 | 726 | 4306 (68,2%) |
| 7,74 | 976,51x | 67,30 | 64,95 | 420 | 5730 (90,8%) |

remoção de 90% dos nós impactou a acurácia em cerca de 5%, reduzindo-a de 67,75% para 62,43%.

Tabela 20 – Avaliação de poda para o conjunto de dados Sintético utilizando 7 árvores com profundidade máxima 7 (Estratégia Flat). Total de Nós: 6748, Acurácia Base: 67,75%.

| **Tempo (s)** | **Speedup** | **Acurácia (%)** | **F1 (%)** | **Chave** | **Nós Removidos** |
|---|---|---|---|---|---|
| 1,30 | 1000,53x | 67,08 | 67,02 | 113 | 493 (7,3%) |
| 1,31 | 992,17x | 65,48 | 65,40 | 426 | 1914 (28,4%) |
| 1,30 | 999,53x | 64,78 | 64,60 | 689 | 3305 (49,0%) |
| 1,31 | 996,40x | 63,86 | 63,71 | 897 | 4687 (69,5%) |
| 1,30 | 997,11x | 62,43 | 62,10 | 655 | 6081 (90,1%) |

Em suma, a implementação em GPU demonstrou acelerações de 900x a 3000x dependendo da densidade do modelo, ilustrativamente, no cenário de maior complexidade (*Covtype*), a busca exaustiva é concluída em apenas 7,7 segundos na GPU, enquanto a projeção baseada no ganho de velocidade indica que o mesmo processo exigiria 6 horas em uma CPU convencional. As estratégias de poda mostraram-se eficazes na redução do tamanho do modelo, embora a estratégia Exponencial tenha demonstrado, em geral, uma capacidade superior de preservação da acurácia em comparação à estratégia *Flat* para taxas de compressão elevadas.

## 4.4 Extensão da Ferramenta TreeLuT

Para a análise do XGBoost, o conjunto de ferramentas Treelut (KHATAEI; BAZARGAN, 2025) foi estendido para permitir a avaliação de dois estilos de implementação de código: uma versão aprimorada baseada em equações e uma nova abordagem baseada em multiplexadores 2:1, como foi detalhado na seção 3.3. Além disso, exploraram-se os hiperparâmetros $w_{feature}$ e $w_{tree}$, variando de um a oito.

Adicionalmente, diferentemente de trabalhos anteriores de FPGA na borda baseados em LUT (KHATAEI; BAZARGAN, 2025; BACELLAR et al., 2024a; ANDRONIC; CONSTANTINIDES, 2024), a abordagem de avaliação proposta verifica também o custo total, incluindo configurações com e sem os módulos de *argmax* e de quantização de entrada para gerar o Min/Max de ajuste do $w_{feature}$. Estes módulos podem aumentar o tamanho total do circuito, mas são necessários em aplicações reais para o processamento de dados brutos. No total, essas combinações resultaram em 1.536 implementações de modelos sintetizadas utilizando a ferramenta Yosys em três conjunto de dados, resumidos na seção 4.1 que avaliaram o custo para diversos valores de $w_{feature}$ e $w_{tree}$ com e sem os módulos adicionais. Todas as implementações foram realizadas utilizando um modelo XGBoost configurado com 7 árvores e uma profundidade máxima de 7.

O espaço total de exploração, totalizando 1.536 implementações distintas de *hardware*, resulta da combinação multiplicativa dos seguintes fatores de projeto:

- **Conjuntos de Dados ($N_{data}$):** 3 conjunto de dados avaliados (*Adult, Covtype e Drybean*).
- **Estilos de Descrição ($N_{style}$):** 2 abordagens de codificação (Multiplexadores vs. Equações).
- **Hiperparâmetros de Quantização ($N_w$):** 64 combinações derivadas da varredura de $w_{feature}$ (8 valores) e $w_{tree}$ (8 valores), ambos no intervalo de $[1, 8]$.
- **Configurações Modulares ($N_{mod}$):** 4 variações topológicas, considerando a presença ou ausência independente dos módulos de *argmax* (2 estados) e de quantização de entrada (2 estados).

Dessa forma, o número total de experimentos ($N_{exp}$) é dado pela Equação 4.4:

$$N_{exp} = 3 \times 2 \times (8 \times 8) \times 4 = 1.536$$

### 4.4.1 Comparação entre implementação das árvores utilizando Multiplexadores e Equações

A ferramenta desenvolvida gera código baseado em multiplexadores que, conforme ilustrado na Figura 22, reduz a contagem de LUTs necessária em comparação com a abordagem baseada em equações do TreeLUT (KHATAEI; BAZARGAN, 2025). Essa redução mantém-se mesmo quando comparada à versão de equação aprimorada, na qual equações longas são decompostas em várias linhas para simplificar a tarefa de síntese. Nos gráficos de área sobrepostos, a implementação “mux” (azul) permanece consistentemente como o limite inferior, superando o estilo “equação” (vermelho) em todas as configurações.

Para cada conjunto de dados, as configurações estão ordenadas pelo seu custo médio de hardware no eixo x, evidenciando uma ampliação na lacuna de recursos à medida que a complexidade do modelo cresce. Além disso, as barras de mínimo-média-máximo na base de cada painel confirmam que a abordagem baseada em mux não apenas reduz a utilização média de recursos em aproximadamente 20%, mas também mantém uma pegada de hardware mais restrita e eficiente quando comparada à alternativa baseada em equações.

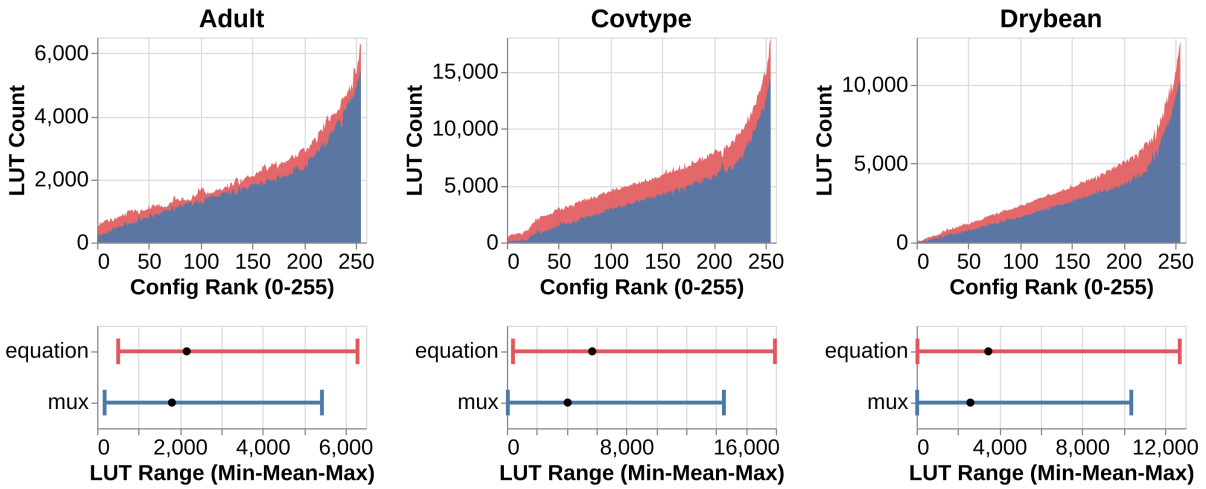



Figura 22 – Comparação de uso de LUTs entre a extensão XGBoost Treelut com descrição Verilog baseada em Mux e em Equação na ferramenta Yosys.

A Tabela 21 detalha as estatísticas de uso de LUTs, confrontando as abordagens. Nota-se que a descrição baseada em multiplexadores apresenta resultados superiores em eficiência de área.

Tabela 21 – Estatísticas detalhadas do uso de LUTs para descrição baseada em Mux e Equação (Total de 1536 configurações).

| | **Adult** | | **Covtype** | | **Drybean** | |
|---|---|---|---|---|---|---|
| **Estatística** | **Mux** | **Eq.** | **Mux** | **Eq.** | **Mux** | **Eq.** |
| Máximo | 5.436 | 6.293 | 14.530 | 17.940 | 10.379 | 12.723 |
| Mediana | 1.572 | 1.780 | 3.660 | 5.264 | 2.151 | 3.000 |
| Mínimo | 181 | 509 | 62 | 410 | 24 | 41 |
| Média | **1.808** | **2.165** | **4.070** | **5.715** | **2.599** | **3.467** |

### 4.4.2 Custo incluindo módulos de Quantização e *Argmax*

Diferentemente de trabalhos anteriores (KHATAEI; BAZARGAN, 2025; WENG et al., 2025; ANDRONIC; CONSTANTINIDES, 2025b; ANDRONIC; CONSTANTINIDES, 2024) que não levam em conta o custo das camadas de quantização e *argmax*, esta seção avaliará essa sobrecarga, uma vez que um hardware completo de inferência na borda será acionado usando dados brutos e deve gerar a classificação final.

A Figura 23 ilustra a variação dos custos de hardware para cada módulo, mensurada pela contagem de LUTs. O eixo x organiza as configurações com base na largura de bits das características ($w_{feature}$), o que gera um gráfico com perfil de "dente de serra". Esse comportamento ocorre porque, para cada $w_{feature}$ fixado, o valor de $w_{tree}$ é incrementado progressivamente. Assim, cada "dente" corresponde a um $w_{feature}$ constante, ao longo do qual $w_{tree}$ escala de 1 a 8 bits. O componente *baseline*, que consiste dos módulos de comparadores, árvores e somas, está representado em azul; em amarelo está destacado o acréscimo do módulo de *argmax*; e em verde o acréscimo do módulo de quantização, que domina os custos adicionais. Portanto, essa disposição resulta em um padrão visual de "dente de serra" composto por oito ondas distintas.

Observa-se que ambas as implementações mantêm curvas de crescimento similares. O acréscimo introduzido pelo módulo *argmax* apresenta-se constante dentro de cada caso de estudo, variando apenas proporcionalmente ao número de classes do conjunto de dados (duas classes para o conjunto "adult" e sete para os demais) e não muda com a variação de $w_{tree}$ ou $w_{feature}$. É importante notar que o crescimento da área ocupada pelo módulo de quantização está estritamente correlacionado ao valor de $w_{feature}$: o impacto é desprezível para $w_{feature} = 1$, mas escala até se tornar o componente dominante do circuito para $w_{feature} = 8$. Esse comportamento deve-se ao crescimento exponencial do número de comparadores necessários para a quantização, dado por $C = 2^{w_{feature}} - 1$.

Como o fluxo de síntese utilizando Yosys (WOLF, 2016) e ABC (BRAYTON; MISHCHENKO, 2010) utiliza a estratégia de *flattening*, o que elimina a hierarquia entre os módulos, ocorre uma otimização global com reaproveitamento das equações Booleanas que compõem as LUTs. Esse fenômeno justifica a anomalia observada para $w_{feature} = 2$, onde o circuito completo (incluindo os módulos de *argmax* e quantização) resultou em uma área menor do que a versão de referência, o que é evidenciado nos valores negativos do gráfico.

Tabela 22 – Uso de LUTs por conjunto de dados e configuração de otimização (Base: Somente as árvores; +Arg: Com Argmax; +Quant: Com Quantização; +All: Argmax e Quantização combinados).

| **Dataset** | $w_{feature}$ | $w_{tree}$ | **Consumo de LUTs** | | | |
|---|---|---|---|---|---|---|
| | | | **Baseline** | **+Arg** | **+Quant** | **+All** |
| *Adult* | 2 | 8 | 1.993 | 1.980 | 1.493 | **1.490** |
| *Dry Bean* | 2 | 8 | 2.451 | 2.738 | 2.052 | **2.336** |
| *Covtype* | 2 | 8 | 5.152 | 5.499 | 4.791 | **5.106** |

A Tabela 22 sintetiza o impacto dos módulos adicionais no consumo de recursos lógicos para esses casos nos quais o número de LUTs é menor acrescentando os módulos de *argmax* e quantização. Nota-se que, embora a implementação isolada do módulo *argmax* possa introduzir um leve custo adicional de área (como visto nos casos *Dry Bean* e

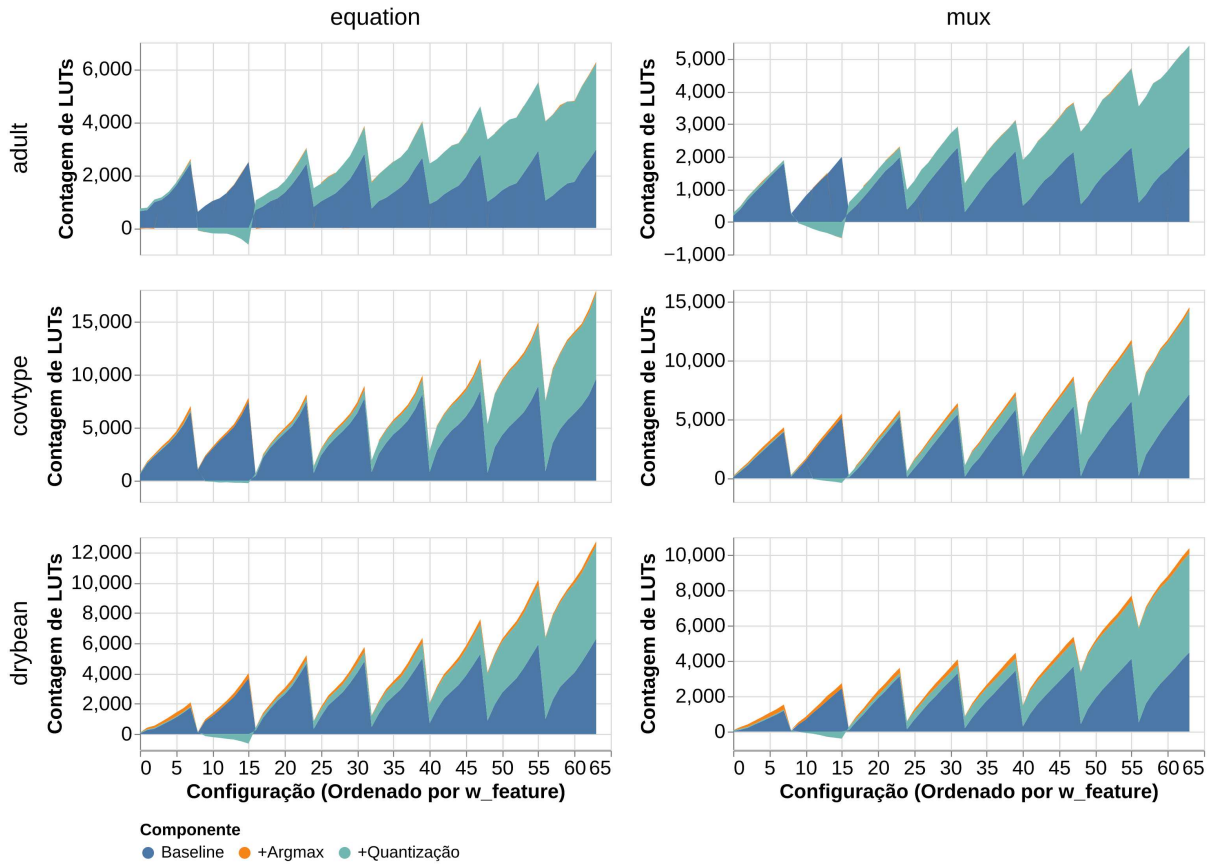



Figura 23 – Contagem média de LUTs para o modelo XGBoost com Treelut, comparando a descrição baseada em mux, incluindo o módulo de saída argmax e módulos de quantização de entrada.

*Covtype*), sua combinação com a quantização resulta em uma redução global do número de LUTs em relação à versão sem os módulos extras para todos os conjuntos de dados. Destaca-se o caso do *Adult*, onde a aplicação conjunta das técnicas reduziu a ocupação de 1.993 para 1.490 LUTs (uma economia de $\approx 25\%$), podendo indicar a existência de um ponto ideal de configuração, a depender da acurácia resultante.

A Tabela 23 sintetiza a variação de LUTs, variando as larguras de bits $w_{feature}$ e $w_{tree}$. Inicialmente, observa-se que a adição do módulo *argmax* (+Arg) sobre a estrutura base das árvores impõe um *overhead* marginal; no caso do *Covtype*, por exemplo, o aumento médio é de apenas 195 LUTs (de 3.771 para 3.966), evidenciando a eficiência desse bloco comparador. Em contrapartida, a configuração completa (+All), que integra o módulo de quantização, apresenta um salto significativo no consumo de área, elevando a média de uso de LUTs em cerca de 50% a 80% dependendo do conjunto de dados.

### 4.4.3 *Trade-offs* entre Acurácia e Área

Nesta seção, será avaliado os *trade-offs* entre a acurácia e área do circuito em função dos hiperparâmetros $w_{feature}$ e $w_{tree}$ nos três conjuntos de dados. Como mencionado, $w_{feature}$ aplica a quantização min-max a cada atributo separadamente; portanto, é possível

Tabela 23 – Estatísticas de uso de LUTs considerando diferentes custos de componentes (Base: Apenas árvores; +Argmax: com argmax; +All: custo total incluindo quantização).

| | **Covtype** | | | **Adult** | | | **Drybean** | | |
|---|---|---|---|---|---|---|---|---|---|
| **Estatística** | **Base** | **+Arg** | **+All** | **Base** | **+Arg** | **+All** | **Base** | **+Arg** | **+All** |
| Máximo | 9.632 | 9.946 | 17.940 | 2.966 | 3.012 | 6.262 | 6.304 | 6.590 | 12.723 |
| Mediana | 3.737 | 3.970 | 5.227 | 1.382 | 1.390 | 2.382 | 2.074 | 2.316 | 3.154 |
| Mínimo | 68 | 62 | 182 | 181 | 185 | 273 | 24 | 25 | 76 |
| Média | **3.771** | **3.966** | **6.009** | **1.419** | **1.429** | **2.552** | **2.204** | **2.404** | **3.860** |

avaliar a sensibilidade da acurácia em função da granularidade da entrada. O parâmetro $w_{tree}$ controla a granularidade da importância do voto de cada árvore. Nos experimentos, utilizaram-se 7 árvores; assim, se $w_{tree} = 1$, cada classe pode receber até 7 votos. Contudo, se $w_{tree} = 3$, cada árvore pode votar de 0 a 7 e, consequentemente, as árvores terão pesos de voto diferentes, podendo cada classe acumular até 49 votos.

Antes de analisar os resultados da Figura 24(a), destacam-se alguns pontos para a correta interpretação visual. Primeiro, a linha azul representa a acurácia (eixo y esquerdo) e a linha vermelha representa as contagens de LUTs, enquanto o eixo x representa o intervalo de $w_{feature}$ de valores de 1 a 8. Segundo, a área sombreada ao redor de cada linha representa a variação em $w_{tree}$ de valores baixos para altos. O objetivo é visualizar o *trade-off* entre redução de LUTs e manutenção da acurácia, para o qual foram adicionadas duas linhas pontilhadas verticais: a vermelha indica o valor pico acurácia, e a azul indica um bom equilíbrio, onde se alcança alta acurácia com baixa contagem de LUTs.

A Figura 24(a) exibe relações distintas entre o parâmetro de quantização $w_{feature}$, a acurácia do modelo e as contagens de LUTs para as configurações sem a quantização e *argmax*, para o estilo mux. O conjunto de dados 'adult' apresenta um platô de desempenho, mantendo acurácia quase constante ( 0,85) para $w_{feature}$ entre 8 e 4. A estreita área sombreada indica robustez contra variações no parâmetro $w_{tree}$. Observa-se que a acurácia cai até 10%, enquanto a contagem de LUTs reduz em 3× para os valores mais baixos. A Figura 24(b) ilustra o impacto do hiperparâmetro $w_{tree}$ no eixo x, onde a área sombreada descreve a sensibilidade a $w_{feature}$.

O conjunto de dados 'drybean' exibe uma relação não monotônica, com pico de acurácia (0,90) no ponto ótimo de $w_{feature} = 5$, seguido por declínio acentuado. Já o conjunto 'covtype' demonstra acentuada insensibilidade a $w_{feature}$, visto que a maioria de seus atributos são binários (44 de 55), e o modelo base possui apenas 7 árvores. Entretanto, a acurácia é afetada pelo hiperparâmetro $w_{tree}$.

Em contraste com as leves variações nos padrões de acurácia, o uso de LUTs exibe uma diminuição notável à medida que $w_{feature}$ e $w_{tree}$ são reduzidos. O parâmetro $w_{feature}$

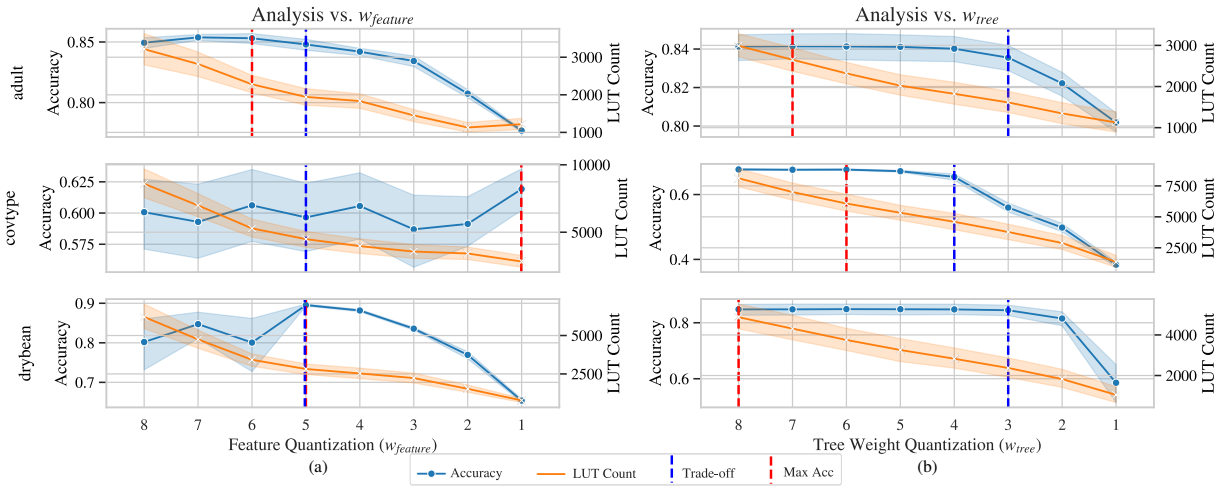



Figura 24 – Acurácia e uso de LUTs em função dos hiperparâmetros do TreeLut $w_{feature}$ e $w_{tree}$.

impacta a largura de bits e o intervalo do comparador, afetando o número de valores distintos de nós da árvore e o tamanho da entrada primária do circuito. Isso gera impacto significativo na otimização da síntese lógica. O parâmetro $w_{tree}$ possui efeito direto na largura de bits da árvore, pois o número de multiplexadores cresce linearmente com a largura de bits. Portanto, a ferramenta de exploração de projeto permite investigar o espaço de *trade-off* entre acurácia e recursos de FPGA para dispositivos de borda.

## 4.5 Go-Fast

Os experimentos da ferramenta *Go-Fast* utilizaram o conjunto de *benchmarks* ISCAS85 (HANSEN; YALCIN; HAYES, 1999), composto por sete circuitos digitais combinacionais de controle, amplamente adotado na literatura como linha de base. A plataforma de hardware utilizada nos testes consiste em um processador Intel i5-12400F (2.5 GHz, 32 GB de RAM) e uma placa gráfica NVIDIA RTX4060 (8GB). O mapeamento dos circuitos para o nível de LUTs foi realizado com as ferramentas Yosys (WOLF, 2016) e ABC (BRAYTON; MISHCHENKO, 2010), executando o *script* de otimização "compress2rs" para gerar redes compostas por LUTs de tamanho máximo 4.

A Tabela 24 apresenta o comparativo de desempenho entre o simulador proposto (*Go-Fast*) e o Verilator (SNYDER, 2018), estado da arte em simulação baseada em CPU. As métricas de vazão detalham o número de amostras simuladas por segundo. A última linha destaca a aceleração média obtida pelas versões do Go-Fast (*multicore* e GPU) em relação ao Verilator.

Para garantir a robustez estatística dos resultados, definiu-se um volume total de simulação, denotado por $|\mathcal{S}|$, de aproximadamente $2^{30}$ vetores de entrada (cerca de 1 bilhão de estímulos). A distribuição dessa carga de trabalho é decomposta conforme a hierarquia de paralelismo de cada arquitetura (GPU ou CPU), conforme as Equações 4.1

Tabela 24 – Tempo de execução e vazão (amostras/s) para benchmarks ISCAS85: Comparativo entre Verilator e Go-Fast (CPU/GPU).

| | | **Go-Fast GPU** | **Vazão (amostras/s)** | | |
|---|---|---|---|---|---|
| **Bench** | **LUTs** | **Tempo (ms)** | **Verilator** | **CPU Go-Fast** | **GPU Go-Fast** |
| c1355 | 86 | 6,47 | $3,26 \times 10^5$ | $1,39 \times 10^9$ | $1,66 \times 10^{11}$ |
| c1908 | 132 | 5,47 | $2,95 \times 10^5$ | $1,33 \times 10^9$ | $1,96 \times 10^{11}$ |
| c3540 | 350 | 9,91 | $1,53 \times 10^5$ | $6,80 \times 10^8$ | $1,08 \times 10^{11}$ |
| c432 | 77 | 4,83 | $5,08 \times 10^5$ | $1,51 \times 10^9$ | $2,22 \times 10^{11}$ |
| c499 | 88 | 6,50 | $3,89 \times 10^5$ | $1,24 \times 10^9$ | $1,65 \times 10^{11}$ |
| c5315 | 486 | 19,48 | $2,18 \times 10^5$ | $2,78 \times 10^8$ | $5,51 \times 10^{10}$ |
| c6288 | 495 | 20,45 | $5,23 \times 10^4$ | $4,36 \times 10^8$ | $5,25 \times 10^{10}$ |
| | | **Speedup Médio** | 1 | $\mathbf{4,15 \times 10^3}$ | $\mathbf{5,72 \times 10^5}$ |

e 4.2:

$$|\mathcal{S}_{GPU}| = \underbrace{N_{grid} \times N_{block}}_{\text{Threads Totais}} \times \underbrace{\eta_{thread}}_{\text{Iterações}} \times \underbrace{\omega_{pack}}_{\text{Bits}} \approx 2^{30} \tag{4.1}$$

$$|\mathcal{S}_{CPU}| = N_{threads} \times N_{batch} \times \omega_{pack} \approx 2^{30} \tag{4.2}$$

Os parâmetros operacionais utilizados para alcançar esse volume estão detalhados na Tabela 25. Na GPU, o paralelismo massivo é explorado através de uma grade ($N_{grid}$) de blocos ($N_{block}$), onde cada *thread* processa uma sequência de $\eta_{thread}$ estímulos. Já na CPU, a carga é dividida entre $N_{threads}$ núcleos físicos, processando lotes grandes ($N_{batch}$). Em ambos os casos, utiliza-se o fator de empacotamento $\omega_{pack} = 32$ para processar múltiplos estímulos em uma única instrução (simulação *bitwise*).

Tabela 25 – Parâmetros de configuração para o volume de simulação $|\mathcal{S}| \approx 2^{30}$.

| **Símbolo** | **Valor** | **Descrição** |
|---|---|---|
| *Configuração GPU* | | |
| $N_{grid}$ | 768 | Número de blocos na Grid |
| $N_{block}$ | 256 | Número de *threads* por Bloco |
| $\eta_{thread}$ | 171 | Estímulos processados por *thread* |
| *Configuração CPU* | | |
| $N_{threads}$ | 12 | *Threads* de CPU concorrentes |
| $N_{batch}$ | $4 \times 10^6$ | Tamanho do lote por *thread* |
| *Configuração Geral* | | |
| $\omega_{pack}$ | 32 | Fator de empacotamento de bits |

Comparando com a literatura, trabalhos anteriores baseados em GPU para Verilog, como (LIN et al., 2022; ELSABBAGH et al., 2023; CHANG; ZHANG; HUANG, 2024),

apresentam acelerações entre duas a quatro ordens de magnitude em relação ao Verilator em núcleo único. O simulador de domínio específico aqui proposto (*Go-Fast*) supera essas marcas, alcançando cinco ordens de magnitude de aceleração, mesmo quando comparado ao Verilator operando em modo *multi-core*.

O tempo médio de execução do Verilator para 1,07 bilhão de estímulos é de aproximadamente uma hora, ao passo que a implementação baseada em GPU requer, em média, 12 milissegundos. O simulador baseado em GPU RTLFlow (LIN et al., 2022), utilizando uma GPU A6000 com 10.572 núcleos, é cerca de 100× mais rápido que o Verilator em uma CPU de 16 núcleos (16th-CPU). O TaroRTL (LIN et al., 2024) utiliza CPU e GPU para melhorar o desempenho do RTLFlow em média 2×. Até mesmo o ASH (ELSABBAGH et al., 2023), um acelerador baseado em *hardware* dedicado para simulação RTL com 256 núcleos, é cerca de 100× mais rápido que o Verilator em CPU de 16 núcleos.

O expressivo ganho de desempenho obtido em relação à CPU justifica-se pela alta eficiência da GPU para esse tipo de problema, viabilizada pelo paralelismo explícito de tarefas independentes e relativamente simples que o hardware gerencia com facilidade. Dessa forma, o módulo gerencia de forma eficiente a execução das simulações, avaliando o impacto de cada variante na qualidade final do circuito.

Tabela 26 – Tempo de execução/vazão para benchmarks EPFL (AMARÚ; GAILLARDON; MICHELI, 2015): Verilator (16K estímulos) vs Go-Fast (2GB estímulos).

| | | **Go-Fast GPU** | **Vazão (amostras/s)** | | |
|---|---|---|---|---|---|
| **Bench** | **LUTs** | **Tempo (ms)** | **Verilator** | **CPU** | **GO-FAST** |
| adder | 250 | 20,67 | $2,54 \times 10^{6}$ | $2,28 \times 10^{8}$ | $5,19 \times 10^{10}$ |
| bar | 896 | 21,79 | $1,59 \times 10^{5}$ | $4,86 \times 10^{8}$ | $4,93 \times 10^{10}$ |
| log2 | 9.172 | 889,69 | $2,08 \times 10^{4}$ | $2,34 \times 10^{7}$ | $1,21 \times 10^{9}$ |
| multiplier | 7.105 | 2.351,06 | $1,61 \times 10^{4}$ | $2,21 \times 10^{7}$ | $4,57 \times 10^{8}$ |
| sin | 1.644 | 39,54 | $4,57 \times 10^{1}$ | $1,52 \times 10^{8}$ | $2,72 \times 10^{10}$ |
| square | 5.532 | 2.719,08 | $2,15 \times 10^{4}$ | $3,86 \times 10^{7}$ | $3,95 \times 10^{8}$ |
| **Média Geométrica do Speedup** | | | 1 | $\mathbf{3,70 \times 10^{3}}$ | $\mathbf{2,20 \times 10^{5}}$ |

Além dos circuitos do ISCAS85, buscamos avaliar a escalabilidade do simulador com circuitos maiores. A Tabela 26 descreve os resultados, considerando o tempo de execução e a vazão em amostras por segundo, para os circuitos combinacionais EPFL (AMARÚ; GAILLARDON; MICHELI, 2015). Estes incluem instâncias variando de 250 até 9.000 LUTs, compatíveis com implementações reais para dispositivos de borda que modelam árvores XGBoost e Redes Neurais utilizando apenas LUTs (BACELLAR et al., 2024b; KHATAEI; BAZARGAN, 2025; GUO, 2025; SUN et al., 2025; ANDRONIC; CONSTANTINIDES, 2025a). Para o Verilator, foram amostrados apenas $2^{14}$ estímulos de entrada devido ao longo tempo de simulação. Em contraste, o Go-Fast simula $2^{30}$ estímulos de

entrada, sendo cinco ordens de magnitude mais rápido que o Verilator.

A avaliação de circuitos em computação aproximada pode basear-se tanto em métricas de classificação quanto de regressão. Para o circuito C880, adotou-se uma métrica de regressão ponderada pela significância dos bits por se tratar de um circuito aritmético com uma saída numérica em 26 bits. Sejam $y$ e $\hat{y}$ os vetores binários de 26 bits representando, respectivamente, o valor obtido e o valor esperado. O erro ponderado $E$ é definido como o somatório das diferenças bit a bit (XOR), escalonadas por um vetor de pesos $w$:

$$E(y, \hat{y}) = \sum_{i=0}^{25} w_i \cdot (y_i \oplus \hat{y}_i) \tag{4.3}$$

Os pesos $w_i$ são atribuídos conforme a posição do bit, onde os 17 bits menos significativos são mascarados (peso zero) e os bits superiores seguem uma progressão geométrica:

$$w_i = \begin{cases} 0 & \text{se } 0 \leq i \leq 16 \\ 2^{i-25} & \text{se } 17 \leq i \leq 25 \end{cases} \tag{4.4}$$

A validação final do circuito é determinada por uma função indicadora, sujeita a um limiar de tolerância $\tau = 0.5$:

$$\text{Valid}(y, \hat{y}) \iff E(y, \hat{y}) \leq 0.5 \tag{4.5}$$

A Figura 25(A) apresenta uma visão simplificada da arquitetura do circuito c880, que implementa uma ALU de 8 bits. Visto que as saídas $O(16:0)$ e $C(8)$ atuam como sinais de controle para módulos externos, elas são irrelevantes para a avaliação do desempenho aritmético. Consequentemente, a atribuição de pesos foca nos bits da saída funcional $F$, conforme ilustrado na Figura 25(B). Nesta configuração, o grupo inicial de 17 saídas recebe peso nulo (mascaramento), enquanto os valores restantes seguem uma curva de crescimento geométrico proporcional à significância do bit, seguindo a equação 4.4.

Com base nesta formulação matemática, a Figura 26 apresenta a distribuição da taxa de erro para 190 mil cenários distintos de poda, gerados pelo simulador Go-Fast. O diagrama de *box-plot* ilustra a relação entre o percentual de remoção de LUTs (eixo horizontal) e a taxa de erro resultante (eixo vertical), exibindo a mediana, os quartis e os valores atípicos.

A análise do gráfico demonstra a dificuldade intrínseca em minimizar o erro à medida que a poda aumenta. A mediana, por exemplo, indica que para 41% de poda, a taxa de erro típica gira em torno de 35%. Contudo, destacam-se os valores atípicos inferiores

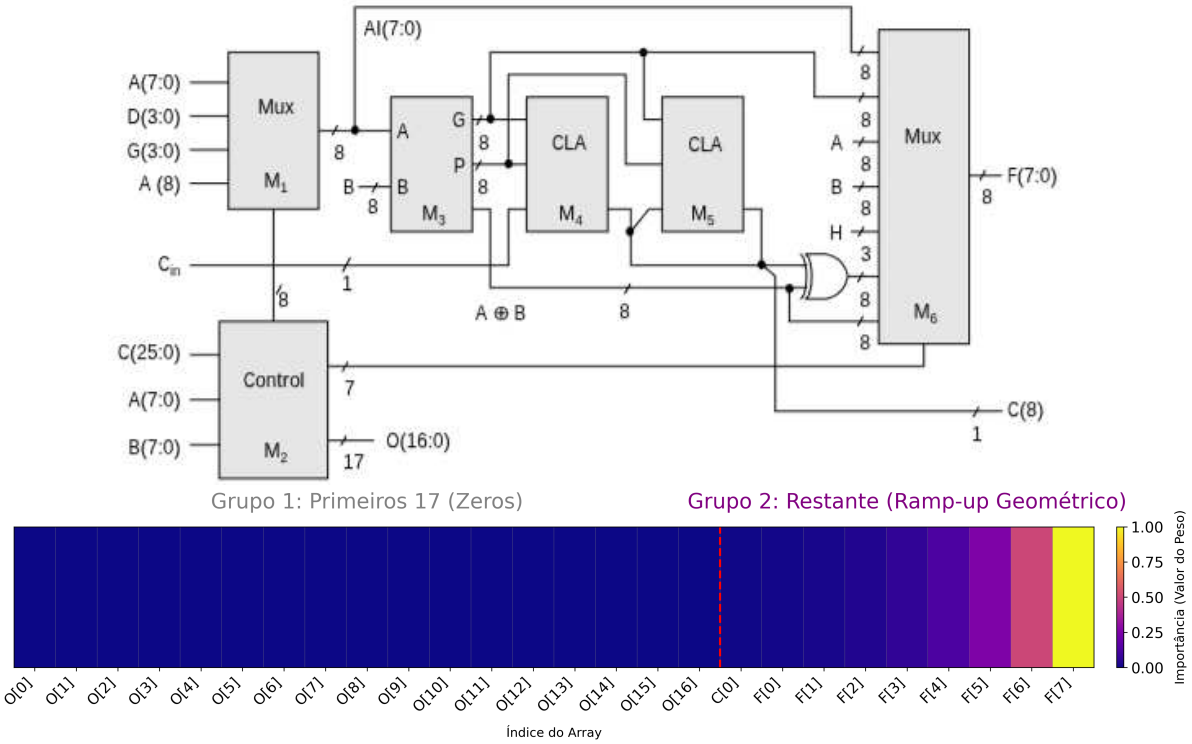



Figura 25 – (A) Arquitetura do circuito c880 (HANSEN; YALCIN; HAYES, 1999) (B) Faixa de pesos $w$ para 26 saídas do círcuito.

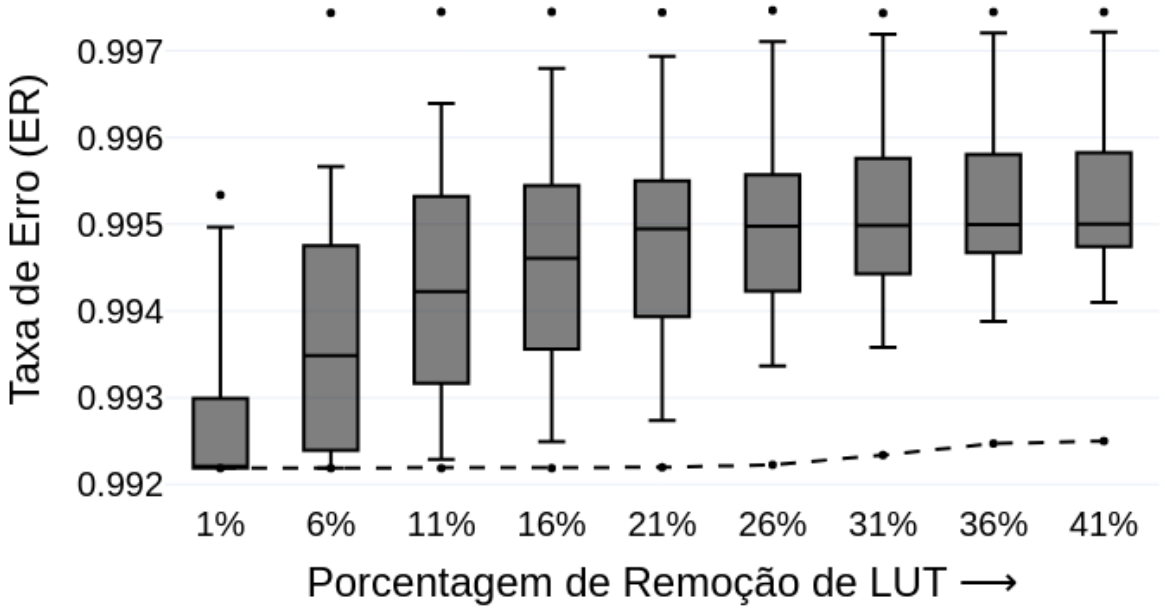



Figura 26 – Remoção de LUTs para c880 utilizando erro de regressão.

(indicados pela linha pontilhada), que representam as melhores configurações encontradas. O caso ótimo alcança uma redução de 41% das 127 LUTs originais, **chegando, então, à 77 LUTs**, mantendo o erro abaixo do limite de 5%.

Esses resultados evidenciam que, embora a maioria das configurações aleatórias degrade severamente o circuito, a exploração massiva permitida pelo desempenho do Go-Fast consegue identificar soluções viáveis de poda substancial. Vale ressaltar, entretanto, que para problemas estritos de classificação, cuja qualquer diferença entre duas saídas constitui um erro, a margem para poda seria drasticamente menor do que a observada nesta métrica de regressão relaxada.

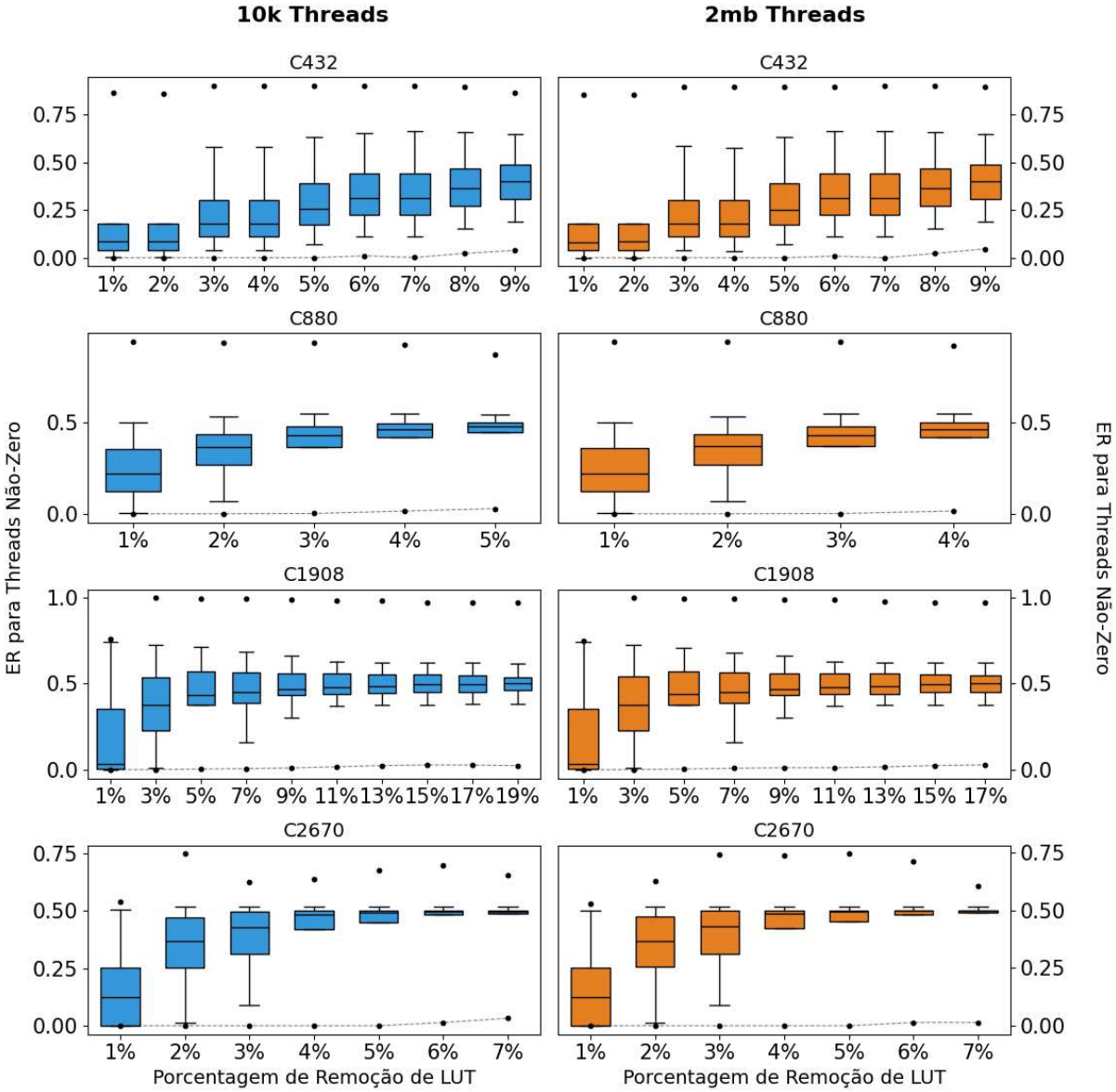



Figura 27 – Taxa de Erro de Classificação vs. Remoção de LUT para 190 mil configurações. A linha tracejada indica a melhor configuração.

A maioria das abordagens de computação aproximada presentes na literatura valida suas funções utilizando conjuntos de estímulos reduzidos, em torno de $2^{10}$ a $2^{13}$ amostras, para minimizar os requisitos de simulação. Contudo, tal prática suscita uma questão crítica: amostras pequenas são suficientes para guiar otimizações de forma confiável ou subestimam as taxas de erro? As capacidades de simulação de alto desempenho do

Go-Fast permitem investigar essa questão através da avaliação de conjuntos de estímulos muito maiores.

A Figura 27 apresenta diagramas de caixa ilustrando o processo de poda de LUTs para quatro circuitos do *benchmark* ISCAS85 (c432, c880, c1908 e c2670) com um limite superior de erro de 5% para uma métrica de classificação. Os gráficos à esquerda mostram taxas de erro calculadas com $2^{21}$ amostras, enquanto os da direita utilizam $2^{14}$ amostras. Essa comparação revela três percepções principais. Primeiro, embora ambos os tamanhos de estímulo produzam formas de distribuição semelhantes para cada circuito, o conjunto menor de estímulos resulta consistentemente em taxas de erro mais baixas, sugerindo que amostras reduzidas podem subestimar o erro real de aproximação. Segundo, o Go-Fast possibilita a validação sobre conjuntos de estímulos significativamente maiores, o que é crucial para uma otimização de poda confiável. Terceiro, a escolha da métrica de erro afeta drasticamente o potencial de poda, como pode ser visto na comparação entre resultados de classificação e regressão para o C880 na Figura 26 mostra que métricas de classificação limitam a redução do circuito a apenas 5%, enquanto métricas de regressão permitem até 40% de redução ao relaxar as restrições de erro.

Tabela 27 – Poda de LUTs usando Go-Fast e ResubALS (MENG et al., 2024).

| | **Número de LUT-4** | | | | |
|---|---|---|---|---|---|
| **Bench** | **Original** | **Go-Fast** | **Resub** | **Go-Fast → Resub** | **Poda Final** |
| c1908 | 132 | 86 | 44 | 43 | 67% |
| c2670 | 290 | 234 | 221 | 216 | 26% |
| c432 | 77 | 49 | 51 | 48 | 38% |
| c880 | 127 | 104 | 87 | 85 | 33% |

Por fim, avaliou-se a poda utilizando uma abordagem ingênua baseada em falhas aleatórias do tipo *stuck-at-0/1*, visando demonstrar a viabilidade de integrar simulação e poda em uma GPU através da incorporação de uma função de controle no *framework* Go-Fast. Compararam-se os resultados de poda com o ResubALS, uma das técnicas ALS do estado da arte (MENG et al., 2024). O ResubALS reduz a complexidade dos nós através da reconstrução aproximada de sua funcionalidade usando uma combinação de nós existentes no circuito. Ao alavancar múltiplos nós para formular funções alternativas, o AppResub alcança maior flexibilidade em comparação com abordagens convencionais, como mudanças aproximadas locais (LACs), resultando tipicamente em erros de aproximação reduzidos.

A Tabela 27 apresenta os resultados obtidos utilizando Go-Fast e ResubALS (MENG et al., 2024). Foram avaliadas três abordagens: Go-Fast, ResubALS e Go-Fast + ResubALS, onde o Go-Fast utiliza poda aleatória ingênua, o ResubALS emprega uma abordagem de reconstrução e o método combinado aplica ambas as técnicas sequencialmente. O limite superior de erro do ResubALS foi fixado em 5% utilizando uma métrica de classi-

ficação. Para os circuitos C2670 e C432, mesmo com uma abordagem de poda ingênua, o Go-Fast alcançou resultados comparáveis ao ResubALS. Para os circuitos C1908 e C880, a aplicação da poda ingênua seguida pelo ResubALS possibilitou uma redução adicional no tamanho do circuito. A redução final de LUTs variou de 26% a 67%.

# 5 Conclusão

Esta dissertação abordou o problema da implementação eficiente de modelos baseados em árvores de decisão (Random Forest e XGBoost) em FPGAs para aplicações de TinyML, desenvolvendo uma metodologia de aplicação que, através de um conjunto integrado de ferramentas, automatiza o fluxo desde o modelo treinado até o circuito otimizado. O trabalho explorou otimizações em múltiplos níveis: poda de árvores, quantização durante o mapeamento e poda de circuitos a nível de LUTs. A seguir, são apresentadas as principais contribuições, limitações identificadas e direções para trabalhos futuros.

## 5.1 Principais Contribuições

As principais contribuições científicas e técnicas deste trabalho são:

**Fluxo de ferramentas integrado:** Desenvolveu-se um conjunto coeso de quatro ferramentas (XGB2GPU, RDSF, TreeLUT e Go-Fast) que cobre o ciclo completo desde o modelo treinado até o circuito otimizado em FPGA. Cada ferramenta foi projetada para atuar em uma etapa específica do processo, mantendo compatibilidade e permitindo a composição de múltiplas otimizações.

**Exploração paralela em GPU:** Demonstrou-se o uso inovador de GPUs para exploração do espaço de projeto em duas etapas críticas: poda de modelos e poda de circuitos. A ferramenta XGB2GPU alcançou speedups de 900× a 3000× para poda de árvores, enquanto a ferramenta Go-Fast obteve acelerações de cinco ordens de magnitude ($10^5$) em comparação com simuladores tradicionais como o Verilator.

**Estratégias de poda topológica:** Propôs-se e avaliou-se três estratégias distintas de poda de árvores (Flat, Linear e Exponencial) que consideram a importância relativa dos nós baseada em sua posição na topologia da árvore. Os resultados experimentais demonstraram que a estratégia Exponencial preserva melhor a acurácia em taxas de compressão elevadas, permitindo redução de até 90% dos nós mantendo degradação de acurácia inferior a 5% em alguns casos.

**Avaliação holística de custos de hardware:** Diferentemente de trabalhos anteriores que consideram apenas o custo das árvores, este trabalho contabiliza o sistema completo incluindo módulos de quantização de entrada e argmax. Esta abordagem fornece uma estimativa mais realista da ocupação de hardware, revelando que para valores altos de $w_{feature}$ (8 bits), o módulo de quantização pode dominar o custo total do circuito, chegando a representar mais área que as próprias árvores.

**Simulação aproximada de alto desempenho:** Demonstrou-se que simulação

com grandes volumes de estímulos ($2^{30}$ amostras) é viável e revela configurações de poda superiores às encontradas com conjuntos reduzidos ($2^{14}$ amostras). Nos experimentos com o circuito C880, a exploração com 194.047 threads identificou 132 soluções com acurácia superior a 60%, enquanto apenas 2 soluções foram encontradas com 1.791 threads, evidenciando a importância da amostragem ampla.

**Validação experimental abrangente:** Realizou-se avaliação em múltiplos conjuntos de dados (Adult, Covtype, Dry Bean e sintéticos) e benchmarks de circuitos (ISCAS85 e EPFL), com análise detalhada de trade-offs entre acurácia, área e diferentes hiperparâmetros de quantização. Os experimentos cobriram 1.536 configurações distintas de hardware para a ferramenta TreeLUT e mais de 190 mil variantes de poda para o Go-Fast.

## 5.2 Validação das Hipóteses

As hipóteses formuladas no Capítulo 1 foram validadas experimentalmente:

**H1 - Poda paralela:** Confirmou-se que a exploração massivamente paralela do espaço de poda utilizando GPUs permite identificar configurações que reduzem significativamente o número de nós nas árvores mantendo degradação de acurácia em níveis aceitáveis. Para o conjunto de dados Adult, alcançou-se redução de 89,5% dos nós com degradação mínima de acurácia (de 84,65% para 84,29%). Para o conjunto de dados sintético com 80% de poda, a melhor configuração encontrada entre 194.047 threads manteve acurácia de 64,59% comparada a 66,21% do modelo original, representando degradação de apenas 1,62%.

**H2 - Quantização e redução de área:** Os experimentos com TreeLUT demonstraram que a quantização dos dados de entrada ($w_{feature}$) e dos pesos das folhas ($w_{tree}$) permite trade-offs favoráveis entre acurácia e área de hardware. Para o conjunto de dados Drybean, observou-se que $w_{feature} = 5$ e $w_{tree} = 3$ alcançam acurácia de 90% com redução de LUTs da ordem de 3× em comparação com configurações de maior precisão. A implementação baseada em multiplexadores reduziu o uso de LUTs em aproximadamente 20% comparada à abordagem por equações.

**H3 - Poda de circuitos com simulação aproximada:** Validou-se que a simulação aproximada acelerada por GPU viabiliza a exploração de estratégias de poda a nível de circuito. Para o circuito C880 com métrica de regressão ponderada, alcançou-se redução de 41% das LUTs mantendo erro inferior a 5%. A comparação com a ferramenta ResubALS demonstrou que a poda ingênua seguida de otimização pode alcançar reduções de área de 26% a 67% nos benchmarks ISCAS85.

**H4 - Ganho de desempenho alcançados:** Confirmaram-se ganhos de desempenho de 3 a 5 ordens de magnitude em comparação com abordagens baseadas em CPU. O

XGB2GPU alcançou acelerações de 900× a 3000× dependendo da densidade do modelo. O Go-Fast obteve um ganho médio de $5,72 \times 10^5$ nos benchmarks ISCAS85 e $2,20 \times 10^5$ nos benchmarks EPFL em comparação com o Verilator multicore.

## 5.3 Limitações

Apesar dos resultados positivos, algumas limitações foram identificadas:

**Restrição a modelos baseados em árvores:** O fluxo de ferramentas desenvolvido é específico para modelos Random Forest e XGBoost. Embora estas sejam arquiteturas importantes para TinyML, outros modelos como redes neurais, SVMs ou k-NN não são suportados. A extensão para outras arquiteturas requereria desenvolvimento de novos módulos de mapeamento e otimização.

**Estratégias de poda heurísticas:** As estratégias de poda implementadas (Flat, Linear e Exponencial) são baseadas em heurísticas topológicas e amostragem estocástica. Embora eficazes, não garantem encontrar a configuração ótima global. Métodos de otimização mais sofisticados, como algoritmos genéticos ou busca bayesiana, poderiam melhorar a qualidade das soluções encontradas.

**Validação limitada a simulação:** A maior parte dos experimentos foi realizada em nível de simulação e síntese lógica, sem validação em hardware físico (síntese place-and-route completa e execução em FPGA real). Embora a contagem de LUTs seja uma métrica consolidada, outros aspectos como temporização, roteamento e consumo energético não foram avaliados empiricamente.

**Análise de consumo energético:** Embora o trabalho foque em redução de área (LUTs) como proxy para eficiência, não foram realizadas medições diretas de consumo energético. Para aplicações de TinyML, onde a energia é crítica, esta seria uma métrica complementar importante.

## 5.4 Trabalhos Futuros

Com base nas contribuições e limitações identificadas, propõem-se as seguintes direções para trabalhos futuros:

**Extensão para outros modelos de aprendizado:** Investigar a aplicabilidade das técnicas de poda e quantização para outros modelos além de árvores de decisão, incluindo redes neurais binárias (BNNs), redes neurais quantizadas (QNNs) e máquinas de vetores de suporte (SVMs). O framework de exploração paralela em GPU poderia ser adaptado para otimizar estes modelos. Comparando-os com os modelos baseados em árvore abortados neste trabalho.

**Integração com ferramentas comerciais de FPGA:** Desenvolver interfaces com ferramentas comerciais como Vivado (Xilinx/AMD) e Quartus (Intel/Altera) para permitir síntese place-and-route completa e validação em hardware real. Isso permitiria avaliar métricas práticas de desempenho, área final após roteamento e consumo energético medido.

**Otimização consciente de aplicação:** Explorar técnicas de poda que considerem as características específicas da aplicação alvo, como distribuição de dados esperada em produção, requisitos de latência e tolerância a erros. Por exemplo, em aplicações de manutenção preditiva, erros em determinadas classes podem ter custo maior que outras.

**Otimização multi-objetivo:** Desenvolver estratégias que otimizem simultaneamente múltiplos objetivos (acurácia, área, latência, consumo energético) utilizando algoritmos de otimização multi-objetivo como NSGA-II ou MOEA/D. Isso permitiria gerar fronteiras de Pareto completas para auxiliar decisões de projeto.

**Aprendizado de métricas de importância:** Investigar o uso de técnicas de aprendizado de máquina para aprender automaticamente as métricas de importância dos nós, substituindo as heurísticas topológicas atuais. Redes neurais poderiam ser treinadas para predizer o impacto da poda de cada nó com base em características estruturais e estatísticas dos dados.

**Validação em aplicações reais de TinyML:** Implementar e validar as ferramentas em aplicações concretas como reconhecimento de atividades humanas com acelerômetros, detecção de anomalias em redes de sensores industriais, classificação de sons ambientais ou diagnóstico médico portátil. Isso forneceria validação empírica da utilidade prática das técnicas propostas.

**Exploração de arquiteturas heterogêneas:** Investigar a partição de modelos entre FPGA e microcontrolador, onde partes do modelo executam em hardware dedicado (FPGA) e outras em software (MCU). Isso poderia combinar as vantagens de eficiência do FPGA com a flexibilidade de programação do MCU.

**Suporte a atualização incremental de modelos:** Explorar técnicas que permitam atualização parcial dos modelos implementados em FPGA sem necessidade de ressintetização completa. Isso seria valioso para cenários onde o modelo precisa ser refinado periodicamente com base em novos dados coletados em campo.

**Análise de robustez e confiabilidade:** Investigar a robustez dos modelos podados frente a variações de dados de entrada, ruído e falhas de hardware. Técnicas de injeção de falhas poderiam avaliar a confiabilidade dos circuitos gerados, aspecto crítico para aplicações de segurança ou missão crítica.

## 5.5 Considerações Finais

Esta dissertação demonstrou a viabilidade de implementar modelos baseados em árvores de decisão em FPGAs para aplicações de TinyML através de um fluxo automatizado de ferramentas que incorpora otimizações em múltiplos níveis. Os resultados experimentais validaram que é possível alcançar reduções substanciais de área de hardware (até 90% via poda de árvores e até 41% via poda de circuitos) mantendo degradação de acurácia controlada.

A exploração massivamente paralela em GPU mostrou-se uma estratégia eficaz tanto para poda de modelos quanto para simulação aproximada de circuitos, alcançando speedups de três a cinco ordens de magnitude em comparação com abordagens tradicionais baseadas em CPU. Estes ganhos de desempenho tornam viável a exploração de espaços de projeto que seriam impraticáveis com ferramentas convencionais.

A principal contribuição deste trabalho reside não apenas nas ferramentas individuais desenvolvidas, mas na demonstração de que otimizações em diferentes níveis de abstração (modelo, mapeamento e circuito) podem ser compostas de forma sinérgica para alcançar implementações eficientes. A contabilização holística dos custos de hardware, incluindo módulos de quantização e argmax, fornece uma visão mais realista da ocupação de recursos em implementações práticas.

As limitações identificadas e os trabalhos futuros propostos indicam que ainda há espaço significativo para avanços nesta área. A validação em hardware real, a extensão para outros modelos de aprendizado e a aplicação em cenários de TinyML de campo são passos naturais para consolidar e expandir as contribuições desta pesquisa.

Em suma, este trabalho contribui para reduzir a lacuna entre as ferramentas de alto nível utilizadas por cientistas de dados e as implementações eficientes em hardware reconfigurável, aproximando a visão de TinyML democratizado onde modelos de aprendizado de máquina podem ser facilmente implantados em dispositivos de borda com recursos limitados.

# Referências


ABADADE, Y. et al. A comprehensive survey on tinyml. **IEEE Access**, IEEE, 2023.

ABDELFATTAH, M. S. et al. Codesign-nas: Automatic fpga/cnn codesign using neural architecture search. In: **Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays**. [S.l.: s.n.], 2020. p. 315–315.

ABDELHADI, A. M.; LEMIEUX, G. G. Modular sram-based binary content-addressable memories. In: **2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines**. [S.l.: s.n.], 2015. p. 207–214.

AGOSTA, G.; BARENGHI, A.; PELOSI, G. Exploiting bit-level parallelism in gpgpus: A case study on keeloq exhaustive key search attack. In: IEEE. **ARCS 2012**. [S.l.], 2012. p. 1–7.

ALCOLEA, A.; RESANO, J. Fpga accelerator for gradient boosting decision trees. **Electronics**, MDPI, v. 10, n. 3, p. 314, 2021.

AMARÚ, L.; GAILLARDON, P.-E.; MICHELI, G. D. The epfl combinational benchmark suite. In: **Proceedings of the 24th International Workshop on Logic & Synthesis (IWLS)**. [S.l.: s.n.], 2015.

ANDRONIC, M.; CONSTANTINIDES, G. A. Neuralut: Hiding neural network density in boolean synthesizable functions. In: IEEE. **2024 34th International Conference on Field-Programmable Logic and Applications (FPL)**. [S.l.], 2024. p. 140–148.

ANDRONIC, M.; CONSTANTINIDES, G. A. Neuralut-assemble: Hardware-aware assembling of sub-neural networks for efficient lut inference. In: IEEE. **IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)**. [S.l.], 2025. p. 208–216.

ANDRONIC, M.; CONSTANTINIDES, G. A. Neuralut-assemble: Hardware-aware assembling of sub-neural networks for efficient lut inference. In: IEEE. **2025 IEEE 33rd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)**. [S.l.], 2025. p. 208–216.

ARAFA, Y. et al. Low overhead instruction latency characterization for nvidia gpgpus. In: IEEE. **2019 IEEE High Performance Extreme Computing Conference (HPEC)**. [S.l.], 2019. p. 1–8.

BACELLAR, A. T. et al. Differentiable weightless neural networks. **arXiv preprint arXiv:2410.11112**, 2024.

BACELLAR, A. T. L. et al. Differentiable weightless neural networks. In: **Proceedings of the 41st International Conference on Machine Learning**. [S.l.]: JMLR.org, 2024. (ICML'24).

BARROS, O. **GitHub Repository for "LoadDataset"**. 2025. Disponível em: <https://github.com/Olavo-B/LoadDatasets>.

BARROS, O. et al. Go fast: Gpu optimizer for fpga-lut approximate simulation and tuning. In: **2025 XV Symposium on Computing Systems Engineering (SBESC)**. [S.l.: s.n.], 2025. p. 1–6.

BARROS, O. A. et al. Rdsf: Everything at same place all at once-a random decision single forest. In: IEEE. **2023 XIII Brazilian Symposium on Computing Systems Engineering (SBESC)**. [S.l.], 2023. p. 1–6.

BECKER, B.; KOHAVI, R. **Adult**. 1996. UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C5XW20.

BLACKARD, J. **Covertype**. 1998. UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C50K5N.

BOUTROS, A.; ARORA, A.; BETZ, V. Field-programmable gate array architecture for deep learning: Survey and future directions. **Proceedings of the IEEE**, IEEE, 2025.

BOUTROS, A.; BETZ, V. Fpga architecture: Principles and progression. **IEEE Circuits and Systems Magazine**, IEEE, v. 21, n. 2, p. 4–29, 2021.

BRAYTON, R.; MISHCHENKO, A. Abc: An academic industrial-strength verification tool. In: SPRINGER. **Computer Aided Verification: 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings 22**. [S.l.], 2010. p. 24–40.

BREIMAN, L. Bagging predictors. **Machine learning**, Springer, v. 24, n. 2, p. 123–140, 1996.

BREIMAN, L. Random forests. **Mach. Learn.**, Kluwer Academic Publishers, USA, v. 45, n. 1, p. 5–32, oct 2001. ISSN 0885-6125. Disponível em: <https://doi.org/10.1023/A:1010933404324>.

CAPOGROSSO, L. et al. A machine learning-oriented survey on tiny machine learning. **IEEE Access**, IEEE, 2024.

CHANG, C.-C.; ZHANG, B.; HUANG, T.-W. Gsap: A gpu-accelerated stochastic graph partitioner. In: **Proceedings of the 53rd International Conference on Parallel Processing**. [S.l.: s.n.], 2024. p. 565–575.

CHEN, T.; GUESTRIN, C. Xgboost: A scalable tree boosting system. In: **Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining**. New York, NY, USA: Association for Computing Machinery, 2016. (KDD '16), p. 785–794. ISBN 9781450342322. Disponível em: <https://doi.org/10.1145/2939672.2939785>.

DONG, X. et al. A survey on ensemble learning. **Frontiers of Computer Science**, Springer, v. 14, p. 241–258, 2020.

DUTTA, L.; BHARALI, S. Tinyml meets iot: A comprehensive survey. **Internet of Things**, Elsevier, v. 16, p. 100461, 2021.

ELSABBAGH, F. et al. Accelerating rtl simulation with hardware-software co-design. In: **Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture**. [S.l.: s.n.], 2023. p. 153–166.

FERREIRA, R. S. **BDBELA: Ferramenta de Síntese Multinível usando BDD.** 1994.

FRANCIS. A tutorial on logic synthesis for lookup-table based fpgas. In: IEEE. **1992 IEEE/ACM International Conference on Computer-Aided Design**. [S.l.], 1992. p. 40–47.

GAJJAR, A. et al. Faxid: Fpga-accelerated xgboost inference for data centers using hls. In: IEEE. **2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)**. [S.l.], 2022. p. 1–9.

GIORDANO, M.; PICCINELLI, L.; MAGNO, M. Survey and comparison of milliwatts micro controllers for tiny machine learning at the edge. In: IEEE. **2022 IEEE 4th International Conference on Artificial Intelligence Circuits and Systems (AICAS)**. [S.l.], 2022. p. 94–97.

GUO, Z. A survey on lut-based deep neural networks implemented in fpgas. **arXiv preprint arXiv:2506.07367**, 2025.

HANSEN, M. C.; YALCIN, H.; HAYES, J. P. Unveiling the iscas-85 benchmarks: A case study in reverse engineering. **IEEE Design & Test of Computers**, IEEE, v. 16, n. 3, p. 72–80, 1999.

IKEDA, T. et al. Hardware/algorithm co-optimization for fully-parallelized compact decision tree ensembles on fpgas. In: SPRINGER. **Applied Reconfigurable Computing. Architectures, Tools, and Applications: 16th International Symposium, ARC 2020, Toledo, Spain, April 1–3, 2020, Proceedings 16**. [S.l.], 2020. p. 345–357.

IMMONEN, R.; HÄMÄLÄINEN, T. Tiny machine learning for resource-constrained microcontrollers. **Journal of Sensors**, Wiley Online Library, v. 2022, n. 1, p. 7437023, 2022.

JARZYNSKI, M.; OLANO, M. Hash functions for gpu rendering. **UMBC Computer Science and Electrical Engineering Department**, 2020.

KALAPOTHAS, S. et al. A survey on risc-v-based machine learning ecosystem. **Information**, MDPI, v. 14, n. 2, p. 64, 2023.

KAVRE, M.; GADEKAR, A.; GADHADE, Y. Internet of things (iot): a survey. In: IEEE. **2019 IEEE pune section international conference (PuneCon)**. [S.l.], 2019. p. 1–6.

KHATAEI, A.; BAZARGAN, K. Treelut: An efficient alternative to deep neural networks for inference acceleration using gradient boosted decision trees. In: **Proceedings of the 2025 ACM/SIGDA International Symposium on Field Programmable Gate Arrays**. [S.l.: s.n.], 2025. p. 14–24.

LIN, D.-L. et al. Tarortl: Accelerating rtl simulation using coroutine-based heterogeneous task graph scheduling. In: SPRINGER. **European Conference on Parallel Processing**. [S.l.], 2024. p. 151–166.

LIN, D.-L. et al. From rtl to cuda: A gpu acceleration flow for rtl simulation with batch stimulus. In: **International Conference on Parallel Processing**. [S.l.: s.n.], 2022. p. 1–12.

LIN, J. et al. Tiny machine learning: Progress and futures [feature]. **IEEE Circuits and Systems Magazine**, IEEE, v. 23, n. 3, p. 8–34, 2023.

LUO, W. et al. Benchmarking and dissecting the nvidia hopper gpu architecture. In: IEEE. **2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)**. [S.l.], 2024. p. 656–667.

MENG, C. et al. Efficient resubstitution-based approximate logic synthesis. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, IEEE, 2024.

MENZE, B. H. et al. A comparison of random forest and its gini importance with standard chemometric methods for the feature selection and classification of spectral data. **BMC bioinformatics**, Springer, v. 10, n. 1, p. 213, 2009.

MISHCHENKO, Y. et al. Low-bit quantization and quantization-aware training for small-footprint keyword spotting. In: IEEE. **2019 18th IEEE International Conference On Machine Learning And Applications (ICMLA)**. [S.l.], 2019. p. 706–711.

NATEKIN, A.; KNOLL, A. Gradient boosting machines, a tutorial. **Frontiers in neurorobotics**, Frontiers Media SA, v. 7, p. 21, 2013.

NVIDIA; VINGELMANN, P.; FITZEK, F. H. **CUDA, release: 10.2.89**. 2020. Disponível em: <https://developer.nvidia.com/cuda-toolkit>.

OWAIDA, M. et al. Scalable inference of decision tree ensembles: Flexible design for cpu-fpga platforms. In: IEEE. **2017 27th International Conference on Field Programmable Logic and Applications (FPL)**. [S.l.], 2017. p. 1–8.

PEDREGOSA, F. et al. Scikit-learn: Machine learning in Python. **Journal of Machine Learning Research**, v. 12, p. 2825–2830, 2011.

SANCHEZ-IBORRA, R.; SKARMETA, A. F. Tinyml-enabled frugal smart objects: Challenges and opportunities. **IEEE Circuits and Systems Magazine**, IEEE, v. 20, n. 3, p. 4–18, 2020.

SAQIB, F. et al. Pipelined decision tree classification accelerator implementation in fpga (dt-caif). **IEEE Transactions on Computers**, v. 64, n. 1, p. 280–285, 2015.

SILVA, A. et al. Implementações eficientes de random forest em fpga de baixo custo para internet das coisas e computação de borda. In: SBC. **Simpósio em Sistemas Computacionais de Alto Desempenho (SSCAD)**. [S.l.], 2024. p. 49–60.

SNYDER, W. Verilator 4.0: open simulation goes multithreaded. In: **Open Source Digital Design Conference (ORConf)**. [S.l.: s.n.], 2018.

SUMMERS, S. et al. Fast inference of boosted decision trees in fpgas for particle physics. **Journal of Instrumentation**, IOP Publishing, v. 15, n. 05, p. P05026, 2020.

SUN, C. et al. da4ml: Distributed arithmetic for real-time neural networks on fpgas. **arXiv preprint arXiv:2507.04535**, 2025.

TABANELLI, E.; TAGLIAVINI, G.; BENINI, L. Dnn is not all you need: Parallelizing non-neural ml algorithms on ultra-low-power iot processors. **ACM Trans. Embed. Comput. Syst.**, Association for Computing Machinery, New York, NY, USA, v. 22, n. 3, abr. 2023. ISSN 1539-9087. Disponível em: <https://doi.org/10.1145/3571133>.

UCI. **Dry Bean**. 2020. UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C50S4B.

VOLKOV, V. Better performance at lower occupancy. In: SAN JOSE, CA. **Proceedings of the GPU technology conference, GTC**. [S.l.], 2010. v. 10, p. 16.

WENG, O. et al. Greater than the sum of its luts: Scaling up lut-based neural networks with amigolut. In: **Proceedings of the 2025 ACM/SIGDA International Symposium on Field Programmable Gate Arrays**. [S.l.: s.n.], 2025. p. 25–35.

WOLF, C. Yosys open synthesis suite. 2016.

YASPR. Disponível em: <https://datafront.maqao.org/public/FPGA/>.