UNIVERSIDADE FEDERAL DE JATAÍ (UFJ)
INSTITUTO DE CIÊNCIAS EXATAS E TECNOLÓGICAS (ICET)
CURSO DE CIÊNCIA DA COMPUTAÇÃO

Luiz Eduardo Rezende Moraes

Inferência de Modelos de Inteligência Artificial na Robotics Language usando TensorFlow Lite Micro para TinyML

Jataí-Goiás

Dezembro de 2025

Luiz Eduardo Rezende Moraes

Inferência de Modelos de Inteligência Artificial na Robotics Language usando TensorFlow Lite Micro para TinyML

Monografia apresentada ao curso de Ciência da Computação do Instituto de Ciências Exatas e Tecnológicas da Universidade Federal de Jataí (UFJ), como requisito para obtenção do título de Bacharel em Ciência da Computação.

Orientador(a):Prof. Dr. Thiago Borges de Oliveira

Jataí-Goiás

Dezembro de 2025

Ficha de identificação da obra elaborada pelo autor, através do Programa de Geração Automática do Sistema de Bibliotecas da UFJ.

Moraes, Luiz Eduardo Rezende

Inferência de Modelos de Inteligência Artificial na Robotics Language usando TensorFlow Lite Micro para TinyML / Luiz Eduardo Rezende Moraes. - 2025.

83 f.: il.

Orientador: Prof. Dr. Thiago Borges de Oliveira

Trabalho de Conclusão de Curso (Graduação) - Universidade Federal de Jataí, Instituto de Ciências Exatas e Tecnológicas, Bacharelado em Ciência da Computação, Jataí, 2025.

Inclui siglas, fotografias, abreviaturas, tabelas, algoritmos, lista de figuras, lista de tabelas, ilustrações, bibliografia, anexos, apêndice.

1. TinyML. 2. Sistemas Embarcados. 3. Robotics Language. 4. TensorFlow Lite Micro. 5. Inteligência Artificial. I. Oliveira, Thiago Borges de , orient. II. Título.

CDU 004

Instituto de Ciências Exatas e Tecnológicas – ICET
Curso de Ciência da Computação

Documento a ser gerado pelo SEI, e incluido como PDF no corpo da monografia.
Informações: Orientando, Título, Orientador, Avaliador 1 e Avaliador 2

Jataí-Goiás
2023

Dedico este trabalho à minha mãe, ao meu pai e ao meu irmão, pelo apoio e incentivo durante minha jornada acadêmica e de vida. Também o dedico a todos que acreditaram em mim e me ajudaram, direta ou indiretamente, a chegar até aqui.

Agradecimentos

Agradeço primeiramente à minha mãe pelo amor, apoio incondicional e sacrifícios durante toda a minha jornada acadêmica. Ao restante da minha família, pelo suporte emocional e incentivo constante. Ao meu professor e orientador Thiago Borges, pela paciência, confiança e ensinamentos que foram muito além do âmbito acadêmico. Aos demais professores, que possibilitaram meu crescimento intelectual necessário para a conclusão deste trabalho. Por fim, agradeço aos meus amigos dentro da faculdade, pelas experiências vividas que tornaram essa jornada mais fácil e divertida, e também aos meus amigos fora da faculdade, que sempre estiveram presentes para me apoiar e distrair quando necessário.

"Nós sempre nos definimos pela capacidade de superar o impossível."
(Cooper em INTERSTELLAR, 2014)

Resumo

O Tiny Machine Learning (TinyML) viabiliza a execução de modelos de aprendizado de máquina em dispositivos de baixo consumo energético, contudo, sua implementação em microcontroladores apresenta desafios significativos devido à complexidade das ferramentas de *software* e à necessidade de conhecimentos avançados em C++ e compilação cruzada. Este trabalho tem como objetivo democratizar o acesso à inteligência artificial embarcada integrando o *framework* TensorFlow Lite Micro (TFLM) à Robotics Language (RL), uma linguagem de programação de propósito específico desenvolvida na Universidade Federal de Jataí com foco na abstração de *hardware* e simplicidade. A metodologia adotada consistiu na implementação de um *wrapper* em C para garantir a interoperabilidade entre a RL e a API do TFLM, no desenvolvimento de uma biblioteca padrão (ai.tflm) e na adaptação experimental do frontend do compilador Robcmp para suporte nativo a novas sintaxes. A validação funcional foi realizada por meio de testes unitários com diversos modelos, incluindo preditores numéricos e classificadores de texto e áudio. Adicionalmente, conduziu-se uma análise comparativa de eficiência entre a solução proposta e implementações nativas em C++. Os resultados demonstraram que a integração é funcional e reduz drasticamente a complexidade do código para o desenvolvedor final, eliminando a necessidade de gerenciamento manual de tensores e operadores. Embora tenha sido observado um leve aumento no tamanho do binário e no tempo de execução devido à camada de abstração, o impacto foi considerado aceitável frente aos ganhos em manutenibilidade e facilidade de uso. Conclui-se que a extensão da RL oferece uma alternativa viável, eficiente e acessível para o desenvolvimento de aplicações de TinyML em sistemas embarcados.

Palavras-chave: *TinyML; Sistemas Embarcados; Robotics Language; TensorFlow Lite Micro; Inteligência Artificial.*

Abstract

Tiny Machine Learning (TinyML) enables the execution of machine learning models on low-power devices; however, its implementation on microcontrollers presents significant challenges due to the complexity of software tools and the need for advanced knowledge in C++ and cross-compilation. This work aims to democratize access to embedded artificial intelligence by integrating the TensorFlow Lite Micro (TFLM) framework into Robotics Language (RL), a domain-specific programming language developed at the Federal University of Jataí focused on hardware abstraction and simplicity. The methodology adopted consisted of implementing a C wrapper to ensure interoperability between RL and the TFLM API, developing a standard library (ai.tflm), and experimentally adapting the Robcmp compiler frontend for native syntax support. Functional validation was performed through unit tests with various models, including numerical predictors and text and audio classifiers. Additionally, a comparative efficiency analysis was conducted between the proposed solution and native C++ implementations. Results demonstrated that the integration is functional and drastically reduces code complexity for the final developer, eliminating the need for manual management of tensors and operators. Although a slight increase in binary size and execution time was observed due to the abstraction layer, the impact was considered acceptable given the gains in maintainability and ease of use. It is concluded that the RL extension offers a viable, efficient, and accessible alternative for developing TinyML applications in embedded systems.

Keywords: *TinyML; Embedded Systems; Robotics Language; TensorFlow Lite Micro; Artificial Intelligence.*

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

AF	Autômato Finito
AIfES	Artificial Intelligence for Embedded Systems
API	Application Programming Interface
ARM	Advanced RISC Machine
AST	Árvore Sintática Abstrata
AVR	Alf and Vegard's RISC processor
CFG	Context-Free Grammar
CPU	Central Processing Unit
DSP	Digital Signal Processor
IA	Inteligência Artificial
IoT	Internet of Things
IR	Intermediate Representation
MCU	Microcontroller Unit
ML	Machine Learning
RAM	Random access memory
RE	Regular Expression
RL	Robotics Language
TFLM	TensorFlow Lite Micro
UFJ	Universidade Federal de Jataí

Sumário

1 Introdução

1.1 Motivação

O Machine Learning (ML) está se tornando cada vez mais comum em diversas áreas da sociedade, como em veículos autônomos, reconhecimento facial e monitoramento de saúde (BIGLARI; TANG, 2023, p. 1). Esse recurso inovador pode estar presente não somente em supercomputadores, celulares e robôs, mas também em sistemas embarcados. Aplicações nessas plataformas já são uma realidade, como apresentado por Hoang, Nam e Park (2019), que implementou um modelo de Inteligência Artificial (IA) para melhorar a identificação de marcações viárias em carros autônomos, e por Attaran et al. (2018), que utilizou circuitos dedicados e plataformas embarcadas para detectar o estresse humano. No entanto, a adoção de ML em microcontroladores de baixo custo ainda é um campo que apresenta desafios significativos, devido a limitações de *hardware* e à complexidade das ferramentas de *software* para o desenvolvimento (SORO, 2021, p. 2, 13).

Ao contrário do que se imagina, microcontroladores (MCU, do inglês *Microcontroller Unit*) podem ser encontrados em diversos aparelhos cotidianos como eletrodomésticos, automóveis, dispositivos de saúde e lâmpadas inteligentes. Qualquer dispositivo que meça, armazene, controle, calcule ou exiba informações é um candidato a ter um microcontrolador embutido (AXELSON, 1997). Eles podem ser definidos como um computador em um único circuito integrado que inclui uma *Central Processing Unit* (CPU), *Random Access Memory* (RAM), alguma forma de *Read-Only Memory* (ROM) e portas de *Input/Output* (I/O) (HUSSAIN et al., 2016). Frequentemente, o termo pode ser utilizado como sinônimo para sistema embarcado, porém existe uma distinção clara entre os dois conceitos. Enquanto o MCU refere-se apenas ao *hardware*, o sistema embarcado compreende o sistema completo (*hardware* + *firmware*), já pronto para uso.

O uso de sistemas embarcados também está fortemente ligado à ideia de *Internet of Things* (IoT), uma rede de dispositivos eletrônicos de baixo custo onde a comunicação e a coleta de dados ocorrem automaticamente, por meio de protocolos de comunicação (WU; QIU; ZHANG, 2020). Atualmente, esse tipo de sistema já é amplamente utilizado para captura e processamento de dados em projetos de IoT como cidades inteligentes, automação residencial e agricultura de precisão (BIGLARI; TANG, 2023). Quando o projeto envolve IA, após a captura dos dados, eles devem ser enviados para essa IA para uma tomada de decisão. Isso pode ocorrer de duas formas: utilizando *Cloud AI* ou *Edge AI*.

Edge AI (Inteligência Artificial na Borda) refere-se à prática de realizar computações de IA próximo aos usuários na borda da rede, em vez de em localizações centralizadas

como os *data centers* de provedores de serviços em nuvem (*Cloud AI*) (SINGH; GILL, 2023). Segundo Singh e Gill (2023), *Edge AI* se destaca em situações que precisam de baixa latência, privacidade e eficiência energética. Já a *Cloud AI* é mais indicada para tarefas que exigem um grande poder de processamento e muito espaço de armazenamento. O uso de ML em MCUs se enquadra no conceito de IA na borda, mas em um campo ainda mais específico e emergente: o *Tiny Machine Learning* (TinyML).

De acordo com Warden e Situnayake (2019), a capacidade de executar um modelo de rede neural com custo energético baixíssimo torna possíveis muitas aplicações completamente novas. Essa é a principal premissa e também definição do termo TinyML, desse modo qualquer aplicação que consiga executar um modelo a um baixo custo de energia pode ser considerada como TinyML. Essa aplicação pode resultar em um produto que é pequeno o suficiente para ser instalado em qualquer ambiente e capaz de operar por um tempo útil sem intervenção humana (WARDEN; SITUNAYAKE, 2019).

Para a execução de TinyML, além dos microcontroladores convencionais, podem ser utilizados outros circuitos integrados (CI), como ASICs (*Application-Specific Integrated Circuits*), FPGAs (*Field-Programmable Gate Arrays*) e DSPs (*Digital Signal Processors*). Apesar da existência de diferentes opções, os MCUs mostram-se mais vantajosos por geralmente serem menores, apresentarem baixíssimo consumo de energia e possuírem custo extremamente baixo. Embora o *hardware* especializado proporcione o melhor desempenho para TinyML, microcontroladores de propósito geral oferecem maior flexibilidade, como discutido por Abadade et al. (2023).

Devido à falta de poder computacional, executar modelos de ML em sistemas embarcados não é uma tarefa fácil; no entanto, segundo Soro (2021), existem três formas principais de realizar esse feito:

- Codificação Manual, implementação manual do modelo de ML diretamente em código de baixo nível, como C ou C++. É uma maneira trabalhosa, demorada e propensa a erros, devido ao grau de complexidade.
- Geração automática de código, uso de ferramentas para converter um modelo pré-treinado em código otimizado para o dispositivo embarcado. A ferramenta converte, por exemplo, um arquivo `.tflite` em código C ou C++.
- Interpretador de ML, é utilizado um interpretador para executar o modelo diretamente no dispositivo.

Várias ferramentas para TinyML já existem no mercado, algumas são produzidas por grandes empresas como a Google, outras são desenvolvidas por grupos menores de desenvolvedores. Cada uma delas possui em sua estrutura uma técnica de geração

automática de código ou um interpretador de ML. O objetivo de um *framework* TinyML é fornecer uma solução abrangente para a construção e implantação de modelos de ML em dispositivos de baixo consumo de energia, facilitando o desenvolvimento de aplicações de *edge computing* por desenvolvedores (ABADADE et al., 2023).

Desenvolvido pela Google, o TensorFlow Lite[1] foi lançado em 2017 e se destacou como um dos pioneiros em *Edge AI*, tendo grande importância na democratização do ML em dispositivos limitados. Atualmente, seu nome foi alterado para LiteRT[2], um produto oficial da Google AI Edge, uma mudança estratégica da empresa para unificar ferramentas de ML no dispositivo. Agora, o LiteRT consegue cobrir múltiplos *frameworks* de treinamento, como JAX[3], Keras[4], PyTorch[5] e TensorFlow. Apesar das atualizações no nome e na amplitude do produto, a ferramenta oficial da Google para desenvolvimento de TinyML em microcontroladores teve seu nome mantido como TensorFlow Lite Micro (TFLM).

Conforme o trabalho de David et al. (2021), o *framework* trabalha com alocação estática de memória, possui um interpretador de ML com menos de 2 kB e já foi validado em diversas arquiteturas, como Arm Cortex-M, ESP32 e DSPs. Sua abordagem baseada em interpretador (que lê o modelo `.tflite` e executa as operações nele contidas) viabiliza a atualização de modelos "em campo", ou seja, em dispositivos que já estão em posse do usuário final ou instalados em seu local de operação. Isso ocorre porque o *firmware* pode ser programado para carregar o modelo de uma área de memória separada, como uma partição Flash, permitindo a substituição do arquivo do modelo sem a necessidade de recompilar o *firmware*.

Além de ser uma área emergente e inovadora dentro do campo de sistemas embarcados e IA, o TinyML possui um grande potencial social e de mercado. Sua capacidade de realizar processamento de dados local com alta privacidade e baixa latência aliada ao baixo custo do *hardware* torna possível a criação de novas aplicações de mercado que antes não eram viáveis, especialmente em áreas remotas ou em dispositivos operados por bateria (SORO, 2021).

Contudo, um dos maiores obstáculos para o aproveitamento desses benefícios, tanto no âmbito educacional quanto para novos desenvolvedores, é a complexidade das ferramentas atuais. A implementação de modelos de TinyML diretamente em C/C++ ou mesmo usando o *framework* TFLM exige conhecimento prévio de C++, CMake e processos de compilação cruzada. Essa barreira técnica limita a experimentação e o desenvolvimento de novas soluções por estudantes, desenvolvedores iniciantes e profissionais que não são

[1] Documentação do TensorFlow Lite: <https://www.tensorflow.org/lite/guide?hl=pt-br>
[2] Documentação do Lite RT: <https://ai.google.dev/edge/litert?hl=pt-br>
[3] Documentação do JAX: <https://docs.jax.dev/en/latest/>
[4] Site do Keras: <https://keras.io>
[5] Site do PyTorch: <https://pytorch.org>

especialistas em sistemas embarcados.

É precisamente nesta lacuna de acessibilidade e simplicidade que este trabalho se insere. Ao unir o TinyML a uma linguagem de programação de alto nível com foco na simplicidade, busca-se abstrair a complexidade inerente ao desenvolvimento de *firmware* e, assim, democratizar o acesso ao desenvolvimento de inteligência artificial embarcada.

No contexto dos sistemas embarcados, o projeto Robotics Language (RL) (PI05974-2024), realizado na Universidade Federal de Jataí (UFJ), desenvolve uma linguagem de programação especializada no campo da robótica e dos microcontroladores. Essa iniciativa é uma evolução de um projeto anterior que lidava com kits robóticos educacionais (PI02361-2018) e busca, de maneira geral, tornar a programação em microcontroladores mais simples e acessível.

Para realizar tal objetivo, a linguagem busca isolar as especificidades de microcontroladores, fornecendo uma camada de abstração de *hardware* dentro do próprio compilador, em vez de depender de *frameworks* ou bibliotecas externas. Essa abordagem permite que os desenvolvedores escrevam o código uma única vez, eliminando a necessidade de adaptar o código para cada *hardware* através de macros condicionais. Ao aproveitar a análise semântica do compilador, é possível prevenir erros comuns encontrados no desenvolvimento de *firmware* em linguagens não específicas de domínio, como C/C++ (OLIVEIRA, 2018).

1.2 Objetivo do Trabalho

Este trabalho teve como objetivo integrar o TFLM à linguagem RL, estendendo sua sintaxe para viabilizar a execução de modelos de ML nos microcontroladores já compatíveis com o *framework* e suportados pela linguagem. Os objetivos específicos foram:

- Adicionar novas palavras-chave, regras sintáticas e semânticas na linguagem RL para a inferência de modelos de TinyML;
- Implementar um *wrapper* parcial da API TFLM em C, para possibilitar a ligação do TFLM com o código compilado em RL;
- Validar a integração por meio de testes funcionais com modelos de IA existentes e já treinados; e
- Realizar uma análise comparativa de eficiência entre a solução em RL e uma implementação nativa em C++.

1.3 Contribuição do Trabalho

A principal contribuição deste trabalho é a extensão da RL para viabilizar o desenvolvimento de aplicações de ML em microcontroladores de forma nativa. Ao integrar o *framework* TFLM, este projeto dota a linguagem de uma sintaxe de alto nível que abstrai a complexidade da inferência. Dessa forma, os desenvolvedores podem aliar o poder do TinyML aos benefícios já oferecidos pela RL, como o desenvolvimento de *firmware* com baixo acoplamento, alta coesão e melhor manutenibilidade.

2 Referencial Teórico

Este capítulo tem como objetivo principal apresentar os conceitos teóricos que são amplamente utilizados ao longo deste trabalho. Ele se mostra fundamental para que seja construída uma base sólida de conhecimentos e conceitos essenciais, os quais são necessários para compreender desde a comparação entre os trabalhos relacionados até a implementação e também a validação da solução proposta.

Inicialmente é apresentada uma visão sobre o funcionamento de um compilador, seguida por uma introdução aos sistemas embarcados e microcontroladores. Em seguida, é abordado o conceito de IA, com foco em TinyML e seus *frameworks*. Por fim, são discutidos os conceitos de interoperabilidade entre linguagens de programação e a RL com seu compilador Robcmp.

2.1 Compilador

Compiladores são programas com uma função aparentemente simples: traduzir código de uma linguagem para outra. No entanto, por trás dessa definição sucinta, está um dos sistemas mais complexos da computação, formado por muitos componentes internos, algoritmos e interações complexas entre eles (COOPER, 2013).

Inicialmente, o compilador pode ser visto como uma caixa-preta que recebe como entrada um código fonte (ou programa fonte) e o transforma em um código objeto (ou programa objeto). A Figura 1 ilustra esse esquema simplificado, que ainda desconsidera as etapas internas do processo de compilação. O código fonte pode ser escrito em diversas linguagens, como C, C++, Fortran, Java ou até mesmo a RL. Já a linguagem do código objeto corresponde ao conjunto de instruções de uma CPU ou MCU específico.

Figura 1 – Esquema simplificado de um compilador. Adaptado de Cooper (2013).

É essencial conhecer as etapas envolvidas no processo de um compilador. As duas principais divisões existentes são: *frontend* e *backend*. Essa divisão de tarefas pode ser visualizada na Figura 2, que sintetiza a estrutura do processo de compilação. O *frontend* do compilador é responsável por analisar o programa fonte, incluindo o processamento léxico,

sintático e semântico, resultando em uma representação intermediária, como uma Árvore Sintática Abstrata (AST, do inglês *Abstract Syntax Tree*) ou um código intermediário. O *backend* recebe essa representação intermediária e gera código alvo otimizado, lidando com detalhes específicos da máquina, como alocação de registradores e seleção de instruções (FISCHER; CYTRON; LEBLANC, 2010).

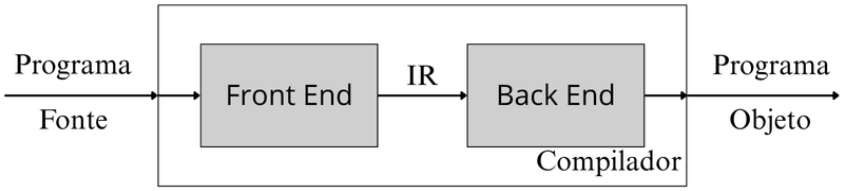

Figura 2 – Compilador dividido em duas etapas. Adaptado de Cooper (2013).

2.2 *Frontend*

Para que o compilador consiga traduzir o código do programa fonte, ele precisa entender tanto a forma (sintaxe) quanto o significado (semântica). O *frontend* é a primeira etapa da compilação e é responsável por determinar se o código está bem construído em termos de sintaxe e semântica. Se ele identificar um código válido, uma representação intermediária é criada; caso contrário, reporta o erro ao usuário para que o problema seja identificado (COOPER, 2013).

A Figura 3 ilustra as principais etapas do *frontend* consideradas neste estudo. Inicialmente, o código-fonte é submetido ao analisador léxico, que converte a sequência de caracteres em *tokens* (unidades significativas). Em seguida, o analisador sintático verifica a estrutura desses *tokens* e gera uma AST. Essa AST é então utilizada pelo analisador semântico para produzir uma Árvore Sintática Decorada, que finalmente é processada para gerar a Representação Intermediária (IR, do inglês *Intermediate Representation*). Todas essas etapas utilizam a tabela de símbolos, seja para armazenar novas entradas ou para verificar a existência de símbolos já declarados.

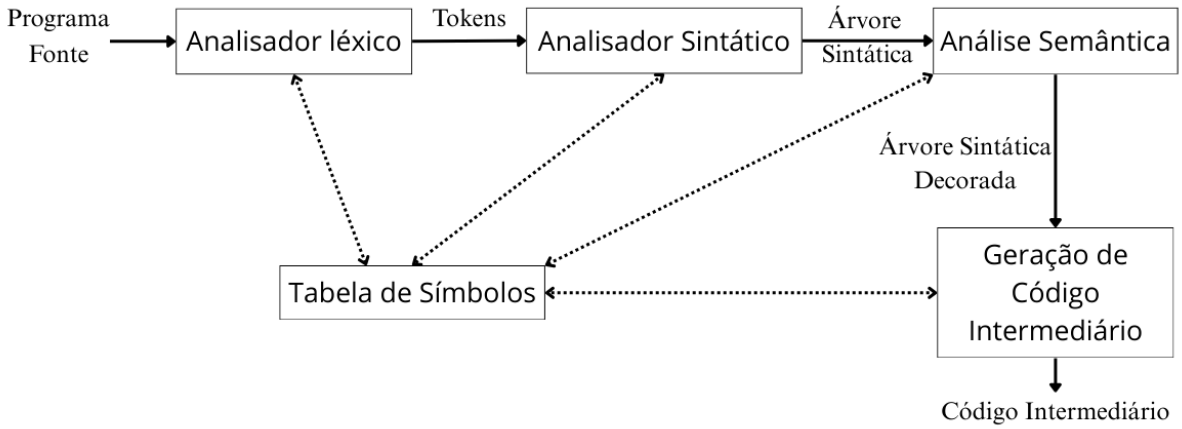

Figura 3 – Etapas do *frontend* de um compilador. Adaptado de Fischer, Cytron e LeBlanc (2010).

2.2.1 Analisador Léxico

A primeira etapa do *frontend* consiste em um analisador léxico, ou *Scanner* em inglês. Ele inicia a análise do programa fonte lendo o texto de entrada (caractere por caractere) e agrupando caracteres individuais em *tokens*, como identificadores, números inteiros, palavras reservadas e delimitadores (FISCHER; CYTRON; LEBLANC, 2010, p. 16). No analisador léxico, Autômatos Finitos (AFs) atuam como núcleo do processo de reconhecimento de *tokens*. Eles simulam um fluxo de transições entre estados, guiados pelos caracteres de entrada, até determinar se uma sequência é válida (aceita) ou inválida (rejeitada) (COOPER, 2013, p. 23).

O conjunto de palavras aceitas por um autômato finito, F, forma uma linguagem, indicada por L(F). Essa linguagem é definida de maneira precisa pelo seu diagrama de transição, que descreve todas as possíveis sequências de estados e símbolos aceitos pelo AF. Para qualquer AF, também podemos descrever sua linguagem usando uma notação chamada expressão regular (RE, do inglês *Regular Expression*). A linguagem descrita por uma RE é chamada de linguagem regular (COOPER, 2013, p. 26). Tanto um AF quanto uma RE podem ser usados em um programa gerador de *scanner*, um programa que produz efetivamente um analisador léxico funcional. Geradores de analisadores léxicos são ferramentas valiosas para a construção de compiladores (FISCHER; CYTRON; LEBLANC, 2010, p. 16).

2.2.2 Analisador Sintático

A principal tarefa do analisador sintático (ou *parser*, em inglês) é verificar se o programa de entrada constitui uma sentença sintaticamente válida na linguagem-fonte. Para essa finalidade, essa etapa utiliza gramáticas livres de contexto, uma vez que REs não são mais suficientes para descrever a sintaxe complexa presente na maioria das linguagens de programação (COOPER, 2013, p. 69, 70).

Apesar de uma RE, como a apresentada na Figura 4, poder reconhecer a expressão $a + b \times c$ corretamente, ela não pode especificar a ordem de precedência entre os operadores. Conforme as regras algébricas convencionais, as operações de multiplicação e divisão possuem precedência sobre adição e subtração. Essa ordem de avaliação pode também ser alterada utilizando parênteses como mecanismo de agrupamento. Porém, a simples inclusão de parênteses em REs não resolve adequadamente o problema de precedência, pois é impossível definir uma RE que reconheça todas as possíveis combinações de parênteses balanceados (COOPER, 2013, p. 71). Em outras palavras, a linguagem reconhecida por uma RE não é suficiente para o reconhecimento sintático do compilador.

$$[a...z]([a...z] \mid [0...9])^* \; ((+ \mid - \mid \times \mid \div) \; [a...z]([a...z] \mid [0...9])^*)^*$$

Figura 4 – RE utilizada para identificar identificadores compostos e operações aritméticas simples entre identificadores, formadas por letras minúsculas, dígitos e operadores (+, -, ×, ÷). Adaptado de Cooper (2013).

Em vez de utilizar REs para verificar sentenças, o analisador sintático emprega uma gramática livre de contexto (CFG, do inglês *Context-Free Grammar*), um conjunto de regras que descreve como as sentenças podem ser formadas. A coleção de todas as sentenças deriváveis a partir de G é denominada linguagem definida por G, representada por L(G) (COOPER, 2013, p. 71). Esse mecanismo demonstra maior poder expressivo e complexidade na especificação de construções sintáticas, uma vez que, diferentemente das REs, possui algumas capacidades adicionais que serão destacadas a seguir.

Uma CFG pode introduzir precedência criando níveis separados na gramática (Expr, Term, Factor). Operadores de maior precedência aparecem em níveis mais internos e, por isso, são reduzidos antes dos de menor precedência.

No exemplo do Código 2.1, a adição e subtração são definidas em expr, enquanto multiplicação e divisão ficam em term e são resolvidas antes de + e - que ficam em expr. A recursão à esquerda (expr → expr op term e term → term op factor) impõe associatividade à esquerda: a - b - c é agrupado como (a - b) - c. Se a gramática tivesse recursão à direita (expr → term ('+' expr | '-' expr)), o agrupamento seria a - (b - c). Parênteses em factor permitem sobrescrever essa ordem.

$$a - b - c \equiv (a - b) - c \qquad a/b/c \equiv (a/b)/c$$

Código 2.1 – Gramática (formato Bison) que define precedência e associatividade para operadores aritméticos

```
expr : expr '+' term
     | expr '-' term
     | term
     ;

term : term '*' factor
     | term '/' factor
     | factor
    ;

factor : '(' expr ')'
       | TOK_IDENT
       | TOK_INTEIRO
       ;
```

Além disso, uma CFG pode validar construções com parênteses balanceados; A regra Expr -> (Expr) na Figura 5 demonstra essa capacidade, permitindo que expressões sejam aninhadas recursivamente dentro de parênteses. O analisador sintático utiliza essa regra para garantir que, para cada parêntese de abertura, exista um correspondente de fechamento.

Por fim, a última capacidade de uma CFG que será abordada é sua representação de hierarquias sintáticas através de árvores de derivação. Na Figura 6, a árvore ilustra a estrutura $(a + b) \times c$. Isso ocorre porque a regra Expr -> Expr Op nome força uma avaliação da associatividade à esquerda. A árvore mostra a operação de adição a + b sendo formada em um nível mais baixo e, em seguida, o resultado dessa subárvore é usado como operando para a multiplicação por c, que está no topo da árvore.

1	*Expr*	→	(*Expr*)
2		|	*Expr Op* nome
3		|	nome
4	*Op*	→	+
5		|	-
6		|	×
7		|	÷

Figura 5 – Regras de derivação de uma CFG. Adaptado de Cooper (2013).

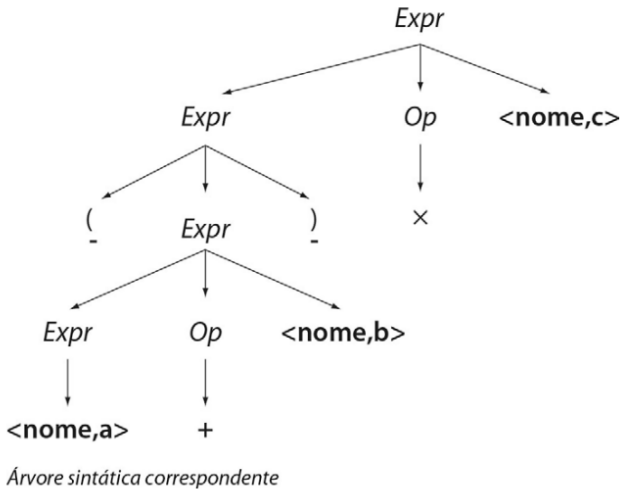

Figura 6 – Árvore sintática de uma CFG. Adaptado de Cooper (2013).

Como resultado da análise sintática, constrói-se uma AST utilizando as informações obtidas do programa fonte e do analisador léxico, assegurando a validação da estrutura sintática do código. Simultaneamente, a tabela de símbolos é atualizada com metadados essenciais sobre cada identificador, incluindo seu tipo de dado, o tamanho de sua representação em memória durante a execução e, no caso de *arrays*, o número de dimensões e os intervalos indexados de cada dimensão (COOPER, 2013, p. 176).

2.2.3 Análise Semântica

A Análise Semântica, como o próprio nome indica, tem como objetivo examinar o significado (semântica) do programa fonte. Para isso, ela utiliza a AST construída pelo Analisador Sintático, juntamente com as informações contidas na tabela de símbolos. Uma das principais funções da análise semântica é a verificação de tipos, na qual o compilador verifica a compatibilidade entre os operandos de cada operador (AHO et al., 2008).

Entre as regras semânticas mais comuns destacam-se:

- A exigência de que índices de *arrays* sejam do tipo inteiro;
- A restrição de que operandos em operações aritméticas possuam tipos compatíveis;
- A proibição de redeclaração de variáveis no mesmo escopo;
- A verificação de que expressões condicionais retornem valores booleanos.

No final dessa fase, temos como resultado a Árvore Sintática Decorada, uma versão aprimorada da AST original. A grande diferença é que agora ela traz consigo uma

informação importante: os tipos de dados de todos os identificadores usados no programa. Essas informações adicionais são vitais para as fases seguintes, especialmente para a geração de código, onde é necessário saber exatamente qual é o tipo de cada variável e expressão.

2.2.4 Geração de Código Intermediário

Durante todo o processo de compilação, podem ser produzidas uma ou mais IRs, as quais podem assumir diversas formas. As árvores sintáticas constituem um tipo de IR, assim como outros formatos de código em baixo nível ou semelhantes à linguagem de máquina. Esses códigos intermediários devem possuir duas propriedades essenciais: precisam ser facilmente gerados e simples de traduzir para a linguagem da máquina alvo (AHO et al., 2008).

O Robcmp utiliza o *backend* da infraestrutura LLVM; por essa razão, sua IR segue o padrão LLVM-IR. O LLVM consiste em uma coleção modular e reutilizável de tecnologias de compilador, que proporciona um otimizador moderno e independente de alvo, além de suporte para a geração de código para diversas CPUs populares.

A representação de código do LLVM foi projetada para ser utilizada de três formas distintas: como uma IR de compilador em memória, como uma representação em bitcode armazenada em disco (adequada para carregamento rápido por um compilador *Just-In-Time*) e como uma linguagem de montagem de alto nível, legível por humanos (The LLVM Project, 2025).

A Figura 7 ilustra uma IR que realiza três operações sequenciais: (1) define uma variável contendo a *string* “hello world”, (2) declara a função externa “puts”, responsável por imprimir *strings* na saída padrão, e (3) implementa a função “main”, que invoca “puts” para exibir a mensagem armazenada.

```
; Declare the string constant as a global constant.
@.str = private unnamed_addr constant [13 x i8] c"hello world\0A\00"

; External declaration of the puts function
declare i32 @puts(ptr captures(none)) nounwind

; Definition of main function
define i32 @main() {
  ; Call puts function to write out the string to stdout.
  call i32 @puts(ptr @.str)
  ret i32 0
}

; Named metadata
!0 = !{i32 42, null, !"string"}
!foo = !{!0}
```

Figura 7 – Exemplo de LLVM-IR de um código “hello world”. Adaptado de The LLVM Project (2025).

2.3 Backend

O papel do *backend* é receber o código IR e produzir um código para a máquina-alvo. Ele seleciona as operações específicas da máquina-alvo para implementar cada operação da IR, determina a ordem mais eficiente de execução dessas operações e decide quais valores serão armazenados em registradores e quais residirão na memória (COOPER, 2013, p. 13). Essas decisões podem ser ajustadas pelo desenvolvedor por meio de flags de otimização, como -Oz (otimização para tamanho extremo) e -Os (otimização para tamanho).

Uma das vantagens do *backend* da LLVM é seu suporte a diversas plataformas, incluindo arquiteturas de CPUs (como x86 e ARM) e famílias de MCUs (como AVR e STM32, ambas baseadas em diferentes arquiteturas). Isso permite que a RL também tenha portabilidade para um amplo espectro de *hardware*. Por exemplo, é possível compilar código para microcontroladores específicos como o ATmega328P (da família AVR, baseada na arquitetura AVR) ou o STM32F103C8T6 (da família STM32, baseada na arquitetura ARM Cortex-M). Estes microcontroladores frequentemente compõem sistemas embarcados, como será detalhado a seguir.

2.4 Sistemas Embarcados

Segundo Marwedel (2021), os sistemas embarcados são sistemas de processamento de informação embutidos nos produtos que os contêm, ou seja, são parte de um sistema maior. Atualmente, estão presentes em inúmeras aplicações, como carros, trens, aviões, equipamentos industriais e de telecomunicações.

Apesar da definição abrangente, é possível obter muitas informações a partir dessa descrição. Como um sistema embarcado faz parte de um sistema maior, ele é frequentemente dedicado a uma tarefa específica. Desse modo, processadores que controlam determinados sistemas de carros ou trens, por exemplo, sempre executam o mesmo *software* (MARWEDEL, 2021, p. 17). Esse *software*, por ser o programa fundamental que gerencia o *hardware* para essas funções específicas, é corretamente classificado como *firmware*.

Além de serem parte fundamental de dispositivos individuais, esses componentes também formam o núcleo de aplicações de IoT. De acordo com Giusto et al. (2010), o termo *Internet of Things* é um paradigma que abrange objetos físicos que se conectam à Internet para compartilhar informações. Tais informações podem ser capturadas por sensores, processadas por um microcontrolador e repassadas a outros dispositivos, a fim de que atinjam um objetivo em comum.

Existe um grande potencial para aplicações de processamento de informações no contexto de IoT e sistemas embarcados. Sua enorme abrangência inclui áreas como eletrônica automotiva, aeronáutica, transporte ferroviário, engenharia marítima, engenharia mecânica, robótica, engenharia civil, recuperação de desastres, engenharia agrícola, setor de saúde, entre outras (MARWEDEL, 2021).

Para que o processamento de informações ocorra, os dados devem ser processados por uma CPU. Nos sistemas embarcados, MCUs são frequentemente empregados para essa tarefa. A CPU integrada a um MCU não só permite executar todas as operações necessárias, como também possibilita que o sistema se beneficie das vantagens inerentes a um microcontrolador.

2.5 Microcontroladores

O núcleo da maioria dos Sistemas Embarcados é o seu centro de processamento, constituído por um MCU. Segundo Hussain et al. (2016), microcontroladores são computadores em um único chip: um circuito que integra uma unidade central de processamento, memória de acesso aleatório, alguma forma de memória apenas de leitura e portas de entrada/saída.

Ao contrário dos computadores de uso geral (como computadores pessoais, *smartphones* e *tablets*), os MCUs são dedicados a tarefas específicas e executam uma única aplicação (HUSSAIN et al., 2016). O MCU é um componente eletrônico único; já os sistemas embarcados são o conjunto completo, que inclui, além do próprio MCU, outros elementos como sensores, atuadores, fontes de alimentação e interfaces de comunicação, todos trabalhando em conjunto para realizar uma função determinada.

Algumas das famílias de microcontroladores mais comuns e que também possuem

microcontroladores suportados pela RL são:

- AVR: É uma família de microcontroladores de 8 bits, baseada na arquitetura AVR, originalmente desenvolvida pela Atmel, empresa fundada em 1984 e especializada em semicondutores para aplicações embarcadas (CORPORATION, 2025). Atualmente, os microcontroladores AVR são produzidos pela Microchip Technology. Estes microcontroladores combinam desempenho eficiente com baixo consumo de energia, oferecendo flexibilidade para uma vasta gama de aplicações embarcadas (Microchip Technology Inc., 2025). A Figura 8 apresenta um exemplo de MCU dessa família.
- STM32: É uma família de microcontroladores de 32 bits fabricada pela STMicroelectronics, baseada na arquitetura ARM Cortex-M. Os MCUs STM32 são otimizados para eficiência energética e operações determinísticas, sendo uma alternativa atraente às arquiteturas de MCU de propósito geral de 8 bits e 16 bits (STMicroelectronics, 2025). A Figura 9 mostra uma placa de desenvolvimento que contém um MCU dessa família.

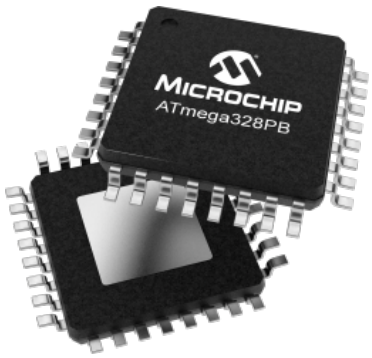

Figura 8 – Microcontrolador de 8 bits ATmega328PB, anteriormente produzido pela Atmel, atualmente fabricado pela Microchip Technology como parte da família AVR. Possui 32KB de memória Flash, 2KB de SRAM e 27 portas de entrada/saída Microchip Technology Inc. (2025).

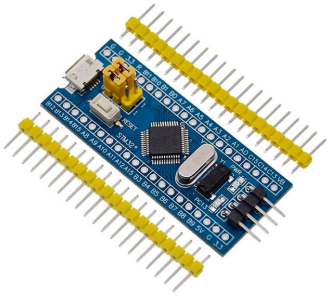

Figura 9 – Placa de desenvolvimento com o MCU STM32F103C8T6, que possui 72 MHz de frequência máxima, 20 kB de memória SRAM, 128 kB de memória FLASH e 37 portas de entrada/saída (STMicroelectronics, 2023).

O avanço dos microcontroladores e das tecnologias de desenvolvimento tem permitido a incorporação de funcionalidades inteligentes diretamente em dispositivos embarcados. Essa evolução cria uma conexão natural com o campo da Inteligência Artificial, tema que será abordado na próxima seção.

2.6 Inteligência Artificial

Na última década, o assunto IA tornou-se popular e impossível de evitar em canais midiáticos como televisão, redes sociais e até mesmo em conversas cotidianas. Pode-se dizer que ele se tornou um termo *mainstream*, expressão do inglês que designa um conteúdo considerado normal que é aceito pela maioria das pessoas (Cambridge University Press, 2025). Essa tecnologia ganhou destaque devido ao seu rápido crescimento e aplicação em diversas áreas da sociedade, consolidando-se como uma inovação promissora que marcou o início de uma nova era a partir da década de 2020.

Apesar de ter se tornado comum e de fácil compreensão em discussões informais, o conceito de IA é frequentemente confundido com ML, como se fossem sinônimos. Segundo Russell e Norvig (2020), o ramo da IA preocupa-se em estudar e construir entidades inteligentes capazes de computar como agir eficientemente em um amplo espectro de situações. Além disso, a IA atualmente abrange diversas subáreas, que vão desde capacidades gerais (como aprendizagem, raciocínio e percepção) até tarefas específicas, tais como jogar xadrez (STOCKFISH, 2025), provar teoremas matemáticos (GAUTHIER; KALISZYK; URBAN, 2022), dirigir veículos (Tesla, Inc., 2025), ou diagnosticar doenças (ATTARAN et al., 2018). Por essa razão, a IA constitui um campo relevante para qualquer tarefa que exija raciocínio intelectual, abrangendo inúmeros domínios do conhecimento.

Enquanto a IA possui um espectro mais abrangente e um objetivo filosófico mais amplo, o ML é um subcampo da IA que estuda como melhorar o desempenho com base na experiência. Ou seja, utiliza dados para realizar um treinamento, permitindo que o sistema computacional tome decisões baseadas em experiências passadas obtidas desses dados. Alguns sistemas de IA empregam métodos de ML, mas outros não (RUSSELL; NORVIG, 2020).

O resultado de treinamentos utilizando grandes bases de dados são modelos de ML, que podem ser executados em diferentes tipos de computadores, dependendo do *hardware* requerido pelo modelo. Esses modelos podem ser executados na borda da rede (*Edge AI*), isto é, em celulares, computadores pessoais e sistemas embarcados, ou na nuvem (*Cloud AI*), isto é, em *data centers* e servidores, para que o modelo possa ser acessado via Internet.

Segundo Singh e Gill (2023), *Edge AI* se destaca em situações que precisam de baixa latência, privacidade e eficiência energética. Já a *Cloud AI* é mais indicada para

tarefas que exigem um grande poder de processamento e muito espaço de armazenamento. O uso de ML em MCUs se enquadra no conceito de IA na Borda, mas em um campo ainda mais específico e emergente: o TinyML, que é o foco principal deste trabalho.

Um exemplo de modelo aplicável a TinyML é o Micro Speech[1], um modelo exemplo do repositório oficial do TFLM. O exemplo Micro Speech demonstra a execução de inferência para reconhecimento de palavras-chave ("sim" e "não"). O sistema consiste em dois modelos principais:

- Pré-processador de áudio: Converte amostras de áudio bruto em dados espectrográficos; e
- Modelo Micro Speech: Modelo compacto (<20 kB) que classifica as palavras-chave a partir dos espectrogramas.

Este projeto irá expandir a RL para suportar a execução de modelos de ML nos MCUs suportados pela linguagem. Dessa forma, será possível realizar inferências com maior privacidade e menor latência, uma vez que não será necessário acessar modelos hospedados em servidores na Internet, mantendo os dados estritamente no âmbito local.

2.6.1 Engines de Inferência

Engines de Inferência (ou Motores de Inferência) são mecanismos que permitem a execução de um modelo de ML já treinado. Na fase de inferência, esse modelo é implementado, e o motor de inferência utiliza a "inteligência" aprendida para compreender e fazer previsões ou classificações sobre dados novos e não vistos.

Geralmente, os mecanismos de inferência fazem parte de um *framework* maior, que contém ferramentas tanto para o treinamento quanto para a inferência de modelos. Entre os exemplos mais famosos estão TensorFlow, PyTorch, Caffe, Keras e JAX.

Esses *frameworks* podem ser usados para treinar desde modelos simples até os mais complexos, mas a escolha do *hardware* para o treinamento depende da necessidade de cada modelo. Enquanto modelos simples podem ser treinados utilizando computadores comuns, modelos mais complexos, como os de *Deep Learning* (Aprendizagem Profunda), são muito mais exigentes e precisam ser treinados em computadores equipados com Unidades de Processamento Gráfico (GPUs, do inglês *Graphics Processing Units*) potentes ou especializadas.

[1] <https://github.com/tensorflow/tflite-micro/tree/main/tensorflow/lite/micro/examples/micro_speech>

2.6.2 Engines para o TinyML

Considera-se TinyML qualquer aplicação capaz de executar modelos de ML com um consumo de energia extremamente baixo (WARDEN; SITUNAYAKE, 2019). Nesse contexto, diferentemente do contexto geral de ML, os *frameworks* disponíveis para o desenvolvimento possuem, na maioria das vezes, somente os mecanismos para inferência. Afinal, o *hardware* de dispositivos como um microcontrolador não é suficiente para o treinamento de um modelo. A fase de treinamento, nesses casos, é feita em uma máquina de desenvolvimento, para que, depois, apenas a inferência seja realizada no dispositivo de borda (SORO, 2021).

Para o desenvolvimento de TinyML em microcontroladores, há diversos *frameworks* disponíveis, incluindo TFLM[2] (Google), uTensor [3] (ARM), MicroTVM[4] (uTVM), STM32Cube.AI[5] (STMicroelectronics), NanoEdge AI Studio[6] (STMicroelectronics), Eloquent TinyML[7], emlearn[8] e EON compiler[9] (Edge Impulse). Embora cada um apresente suas particularidades e suporte a diferentes famílias de MCUs, o TFLM foi selecionado para este trabalho por suas vantagens e conveniência, alinhando-se aos interesses do projeto.

Dentre as conveniências estão o suporte para microcontroladores da família STM32 (baseada na arquitetura ARM Cortex-M) tanto pelo Robcmp quanto pelo TFLM, a existência de modelos já treinados em formato `.tflite` que podem ser aproveitados, e uma licença que permite uso gratuito, modificação e distribuição, a qual pode ser encontrada no repositório do GitHub, que aponta para a licença Apache License, Version 2.0[10]. Dentre os interesses do projeto estão: manter o código aberto, buscar expandir para outras plataformas futuramente e o foco em portabilidade e suporte a *hardware* heterogêneo.

2.6.3 TensorFlow Lite Micro

Conforme David et al. (2021), o TFLM possui uma série de vantagens que podem ser observadas após a análise das decisões de design e implementação. Dentre essas vantagens, as mais pertinentes e em conformidade com o objetivo da RL são:

[2] Documentação do TFLM: <https://github.com/tensorflow/tflite-micro>
[3] Documentação do uTensor: <https://github.com/uTensor/uTensor>
[4] Documentação do MicroTVM: <https://daobook.github.io/tvm/docs/topic/microtvm/index.html>
[5] Documentação do STM32Cube.AI: <https://stm32ai.st.com/stm32-cube-ai/>
[6] Site do NanoEdge AI Studio: <https://stm32ai.st.com/nanoedge-ai/>
[7] Documentação do Eloquent TinyML: <https://github.com/eloquentarduino/EloquentTinyML>
[8] Documentação do emlearn: <https://github.com/emlearn/emlearn>
[9] Documentação do EON Compiler: <https://docs.edgeimpulse.com/docs/edge-impulse-studio/deployment/eon-compiler>
[10] Disponível em: <https://www.apache.org/licenses/LICENSE-2.0>

- Código Aberto: TFLM é um *framework* de inferência TinyML de código aberto disponível no GitHub.
- Abordagem Baseada em Interpretador: Utiliza um interpretador que oferece flexibilidade e portabilidade, facilitando a adaptação a novas aplicações e recursos. Diferente da geração de código, o interpretador permite atualizar modelos substituindo apenas o arquivo/área de memória do modelo, sem recompilar tudo.
- Independência de *hardware*: Minimiza o uso de dependências externas e requisitos de biblioteca para ser independente em relação ao *hardware*.
- Gerenciamento de Memória Eficiente: Não depende de alocação dinâmica; utiliza uma "arena" de memória fornecida pela aplicação.
- Reutilização de Ferramentas TensorFlow: Integra-se fortemente com o ambiente de treinamento do TensorFlow e reutiliza as ferramentas e *kernels* de referência do TensorFlow Lite, facilitando a conversão, otimização e garantindo um ambiente harmonizado.
- Suporte a Otimizações: Suporta quantização (como 8 bits) e outras otimizações (*folding*, remoção de *dropout*) através da *toolchain* do TensorFlow Lite.

2.7 Interoperabilidade entre linguagens de programação

Conforme International Organization for Standardization (ISO), International Electrotechnical Commission (IEC) e Institute of Electrical and Electronics Engineers (IEEE) (2017), interoperabilidade pode ser definida como "O grau em que dois ou mais sistemas, produtos ou componentes podem trocar informações e usar tais informações". Em essência, a interoperabilidade é a capacidade de diferentes sistemas ou componentes de *software* trocarem informações e utilizarem as funcionalidades uns dos outros, mesmo que tenham sido desenvolvidos com tecnologias, plataformas ou linguagens de programação distintas.

A interoperabilidade pode ser alcançada de diferentes maneiras, dependendo das tecnologias ou linguagens de programação utilizadas no projeto. Ela pode ocorrer entre diferentes linguagens de programação, por exemplo, C e Python, por meio de módulos existentes, como ctypes[11] e cffi[12]. Nesses casos, a interoperabilidade pode ser interpretada como Interface de Função Estrangeira (FFI, do inglês *Foreign Function Interface*).

No contexto deste projeto, a RL será integrada ao TFLM. Para que a RL invoque as funções de inferência disponibilizadas pela API em C++, será criada uma API intermediária

[11] Disponível em: <https://docs.python.org/3/library/ctypes.html>
[12] Disponível em: <https://cffi.readthedocs.io/en/latest/>

(*wrapper*) implementada em C++, mas exposta por meio de uma interface em C. Isso permitirá a interoperabilidade entre os dois componentes em tempo de ligação (*link-time*); assim, o código em RL, compilado com o robcmp, poderá ser ligado ao código C++ do TFLM, compilado com um compilador padrão C++.

2.8 Robotics Language and Compiler

Desenvolvidos na UFJ em 2018, os trabalhos de Sousa (2018) e Rodrigues (2018) integraram o projeto "Especificação e Construção de Protótipos Funcionais de Kits Robóticos de Baixo Custo para uso em Processos de Ensino-Aprendizagem" (PI02361-2018), que visava criar um ecossistema de robótica educacional de baixo custo. Sousa dedicou-se à especificação do *hardware*, avaliando componentes por preço e desempenho, enquanto Rodrigues criou o *software*, desenvolvendo a Linguagem para Robótica Educacional (LRE) em português e seu respectivo compilador para facilitar a programação. Posteriormente, Subhi (2019) realizou adaptações e melhorias na linguagem, como a mudança da língua para o inglês e a adição de funcionalidades como funções com parâmetros, vetores e matrizes estáticas.

Posteriormente, surgiu o projeto denominado "Robotics Language: Uma Linguagem de Programação de Propósito Específico para Microcontroladores" (PI05974-2024). Trabalhos relacionados a esse projeto implementam funcionalidades até então inexistentes na biblioteca padrão da linguagem. Por exemplo, Santos (2024) implementou funções matemáticas, como seno e cosseno. Já Vieira (2024) adicionou funções para manipulação de *strings*.

A RL busca isolar as especificidades de microcontroladores, fornecendo uma camada de abstração de *hardware* dentro do próprio compilador, em vez de depender de *frameworks* ou bibliotecas externas. Essa abordagem permite que os desenvolvedores escrevam o código uma única vez, eliminando a necessidade de adaptar o código para cada *hardware* através de macros condicionais. Ao aproveitar a análise semântica do compilador, é possível prevenir erros comuns encontrados no desenvolvimento de *firmware* em linguagens não específicas de domínio, como C/C++ (OLIVEIRA, 2018). Além disso, ela possui suporte a interfaces, tipos complexos (com variáveis e métodos, similar a classes da orientação a objetos), além de um sistema de tipagem implícito.

Seu compilador, o Robcmp, foi escrito em C++ e utilizou ferramentas como o Flex[13] (versão 2.6.4) para análise léxica e o Bison[14] (versão 3.8.2) para análise sintática. Para a interpretação da representação intermediária, foi utilizado o *backend* oferecido pelo LLVM. Desde o começo do projeto, a versão do LLVM utilizada é constantemente

[13] Disponível em: <https://github.com/westes/flex>
[14] Disponível em: <https://www.gnu.org/software/bison>

atualizada, para que o compilador se mantenha sempre atualizado.

Para o desenvolvimento de programas e *firmwares* na Robotics Language é recomendado o uso do editor de texto e IDE (*Integrated Development Environment*) Visual Studio Code[15], devido ao seu fácil acesso a extensões. A principal extensão necessária para o desenvolvimento é a PlatformIO[16]. Ela é responsável pela execução, depuração e carregamento dos *firmwares*. Outra extensão essencial é a RobCmpSyntax, criada com o intuito de ajudar o programador a visualizar a estrutura de seu código a partir de cores diferentes para a sintaxe.

[15] Disponível em: <https://code.visualstudio.com>
[16] Disponível em: <https://platformio.org/>

3 Trabalhos relacionados

Neste capítulo, são apresentadas publicações relacionadas a este trabalho, abordando decisões de design de ferramentas para TinyML e *Edge AI* no contexto de MCUs. A busca dos trabalhos foi realizada por meio de uma revisão narrativa de literatura (RNL), utilizando como ponto inicial o conjunto de trabalhos já defendidos dentro do projeto Robcmp (SANTOS, 2024; VIEIRA, 2024; SUBHI, 2019), seguido pela busca de trabalhos correlatos no Google Scholar. Foram utilizados termos como "TinyML frameworks", "Machine Learning on Microcontrollers", "Embedded Machine Learning" e "Edge AI frameworks".

3.1 Metodologia de análise

Para garantir uma avaliação estruturada dos trabalhos selecionados, foram definidos quatro critérios principais de análise: suporte a diferentes MCUs (SUP), facilidade de uso e documentação (USO), recursos de otimização (OPT) e suporte a treinamento e embarcado (TRE). Esses critérios avaliam se o trabalho/ferramenta possui grande aplicabilidade, é acessível para desenvolvedores, apresenta técnicas eficazes para otimização de modelos e se é inovador e oferece suporte ao treinamento diretamente no dispositivo embarcado.

3.1.1 Suporte a diferentes MCUs (SUP)

Avalia se a ferramenta ou *framework* oferece compatibilidade com múltiplas famílias de microcontroladores, como STM32, ESP32, AVR, PIC, MSP430, Renesas RX, NXP LPC, entre outras, ampliando sua aplicabilidade em projetos embarcados.

3.1.2 Facilidade de uso e documentação (USO)

Considera a existência de documentação clara, exemplos práticos e facilidade de integração, aspectos essenciais para adoção por desenvolvedores.

3.1.3 Recursos de otimização (OPT)

Analisa se o trabalho apresenta técnicas para otimização de modelos, como quantização, compressão ou aceleração por *hardware*, visando eficiência em dispositivos com

recursos limitados.

3.1.4 Suporte a treinamento e embarcado (TRE)

Verifica se a ferramenta permite tanto o treinamento quanto a inferência diretamente no dispositivo embarcado, ampliando as possibilidades de uso em aplicações reais.

3.2 Trabalhos analisados

Foram analisados três trabalhos que abordam ferramentas para implementação de modelos de ML em MCUs, cada um com diferentes abordagens e características. Dentre eles, o primeiro foi selecionado por ser a principal base do trabalho proposto e o segundo por ser uma ferramenta amplamente utilizada no desenvolvimento embarcado com MCUs STM32. Por fim, o terceiro trabalho foi selecionado por se tratar de uma ferramenta recém-proposta (publicada no IEEE em 2024) e desenvolvida por um grupo de pesquisadores, ao invés de uma grande empresa como o Google ou a STMicroelectronics. Isso adiciona uma perspectiva acadêmica distinta ao desenvolvimento de ferramentas de ML para MCUs.

3.2.1 TensorFlow Lite Micro: Embedded Machine Learning for TinyML Systems

O trabalho de David et al. (2021) apresenta as principais decisões por trás do TFLM, o *framework* de inferência de ML da Google escolhido para este projeto. Seu grande diferencial é a abordagem baseada em um interpretador, que oferece flexibilidade e portabilidade para lidar com a fragmentação do ecossistema de *hardware* embarcado.

Em relação ao seu suporte a diferentes MCUs, o TFLM foi projetado para ser altamente portável e agnóstico em relação ao *hardware*, tendo sido validado em diversas arquiteturas, como ARM Cortex-M, ESP32 e vários DSPs. A especialização para plataformas específicas é alcançada permitindo que fabricantes de *hardware* contribuam com *kernels* de operadores otimizados (por exemplo, a biblioteca CMSIS-NN da Arm), que podem ser integrados de forma modular. Esse tipo de biblioteca implementa os operadores de ML mais comuns, como convoluções e funções de ativação, aproveitando as instruções específicas do processador para melhorar o desempenho. Dessa forma, os *kernels* comuns de TinyML, como ADD, GATHER, SOFTMAX, CONV2D, FULLYCONNECTED, MAXPOOL2D, entre outros, são executados de forma otimizada em processadores ARM Cortex-M.

Quanto à facilidade de uso e documentação, o TFLM se integra fortemente ao ecossistema *TensorFlow*, reutilizando suas ferramentas para conversão e otimização de

modelos, o que facilita a adoção por desenvolvedores familiarizados com *TensorFlow*. Porém, seu fluxo de desenvolvimento pode ser um pouco complexo para iniciantes, exigindo conhecimento prévio de C++, *CMake*, compilação cruzada e alta familiaridade com o modelo de ML que será utilizado. A documentação oficial é abrangente, incluindo guias de início rápido, tutoriais e exemplos de código, o que ajuda a mitigar minimamente a curva de aprendizado.

O *framework* também oferece algumas técnicas de otimização herdadas do *TensorFlow Lite*, como a quantização de modelos para 8 bits, que reduz o tamanho do modelo e o consumo de memória. Além disso, inclui um "Planejador de Memória" que aplica um algoritmo de *bin-packing* para reutilizar buffers de tensores intermediários, reduzindo substancialmente o pico de uso de RAM. Em vez de alocar memória para todos os tensores intermediários que um modelo precisa, esse algoritmo consegue reutilizar buffers de tensores que não são mais necessários. Ademais, o uso de *kernels* otimizados, como o CMSIS-NN, demonstrou melhorias de desempenho superiores a 4x em comparação com as implementações de referência.

Como o TFLM é um *framework* de inferência, ele não oferece suporte ao treinamento de modelos diretamente no dispositivo embarcado. O fluxo recomendado envolve o treinamento do modelo em um ambiente mais poderoso, seguido pela conversão do modelo treinado para o formato `.tflite` e sua implantação no dispositivo embarcado para inferência.

3.2.2 STM32Cube.AI

O STM32Cube.AI, detalhado no manual do usuário da STMicroelectronics (2021), é uma ferramenta proprietária desenvolvida pela STMicroelectronics, projetada para facilitar a implementação de modelos de inteligência artificial em microcontroladores STM32. Ele se integra ao ecossistema STM32Cube, que inclui uma série de ferramentas e bibliotecas para desenvolvimento embarcado. Diferente do TFLM, que utiliza um interpretador, essa ferramenta converte modelos pré-treinados diretamente em uma biblioteca C, otimizada e específica para a plataforma alvo.

Em relação ao suporte a diferentes MCUs, o STM32Cube.AI é especificamente projetado para a família de microcontroladores STM32 da STMicroelectronics, mostrando-se uma solução mais específica e menos flexível em comparação com o TFLM e AIfES. Dentro da família STM32, ele suporta uma ampla gama de séries como STM32F4, STM32F7, STM32H7, STM32L4, entre outras.

A ferramenta é integrada ao ambiente de desenvolvimento STM32CubeMX, oferecendo uma interface gráfica que guia o usuário em todas as etapas do processo. O fluxo de

trabalho consiste em selecionar o MCU, carregar o modelo pré-treinado e a ferramenta gera automaticamente um projeto C completo e pronto para compilação. Na questão de documentação, o próprio manual do usuário é a principal referência, sendo bastante detalhado e complementado por documentações adicionais incluídas no pacote de *software*.

A otimização é o principal ponto forte do STM32Cube.AI. A abordagem de geração de código mostra-se mais eficaz que o uso de um interpretador, pois elimina o *overhead* associado à interpretação em tempo de execução, como afirma o manual. A ferramenta emprega várias técnicas, como a fusão de operações (*operation fusing*), que combina camadas (por exemplo, uma ativação após uma convolução) para otimizar o uso da memória e da computação. Além disso, ela também suporta quantização de modelos para 8 bits e a compressão de pesos para camadas densas, visando reduzir o uso de memória Flash.

Como essa é uma ferramenta focada em inferência, o STM32Cube.AI não oferece suporte ao treinamento de modelos diretamente no dispositivo embarcado. O fluxo recomendado é a utilização de um modelo pré-treinado nos formatos *ONNX*, *Keras* ou *TensorFlow Lite* para a conversão em um projeto C.

3.2.3 AIfES: A Next-Generation Edge AI Framework

O trabalho de Wulfert et al. (2024) apresenta o *Artificial Intelligence for Embedded Systems* (AIfES), um *framework* escrito em C de código aberto para *Edge AI*, tanto para o treinamento quanto para a inferência de modelos. Ele surge como uma alternativa ao TFLM, com foco em ter uma arquitetura de *software* altamente modular, projetada para facilitar a integração de otimizações e aceleradores de *hardware* customizados.

Por ser um *framework* escrito em C e compatível com o compilador GCC, ele pode ser executado em qualquer *hardware* suportado por essa ferramenta, abrangendo desde MCUs de 8 bits até arquiteturas mais potentes como ARM Cortex-M. Sua principal vantagem nesse quesito é sua arquitetura modular, que permite que o desenvolvedor adicione otimizações para plataformas específicas, como a biblioteca CMSIS para MCUs ARM, através de módulos de implementação dedicados.

Quanto à facilidade de uso, o *framework* busca ser intuitivo no quesito treinamento, e possui um fluxo de desenvolvimento similar aos *frameworks* de treinamento populares como *PyTorch* e *Keras*, onde o modelo é construído a partir de camadas (*layers*), funções de perda (*loss*) e otimizadores (*optimizers*).

Por outro lado, o fluxo de desenvolvimento para a inferência é mais complexo. O AIfES difere de abordagens que utilizam um único arquivo de modelo, como o TFLM. O processo de conversão exige que o desenvolvedor primeiro recrie a arquitetura da rede neural

diretamente no código C, utilizando as funções e camadas/*kernels* fornecidos pelo AIfES. Em seguida, os pesos treinados devem ser exportados de um ambiente de treinamento (como *Python*) e carregados manualmente no código do AIfES. Esse processo pode ser mais trabalhoso e propenso a erros, especialmente para modelos complexos. A documentação oficial detalhada tanto para o treinamento quanto para a inferência está disponível no repositório do GitHub[1].

No quesito de otimização, o AIfES implementa diversas estratégias. Para alocação de memória, ele utiliza um alocador estático que calcula previamente e distribui um único bloco de memória para o modelo, evitando o uso de alocação dinâmica e a ocorrência de fragmentação. Para a otimização do treinamento, o *framework* implementa um fluxo de retropropagação de baixo consumo de memória, conhecido como *Lightweight Backpropagation*, que aumenta a eficiência no uso da RAM. Além disso, ele também suporta quantização inteira simétrica de 8 bits e 32 bits e permite a integração de bibliotecas otimizadas como já mencionado anteriormente.

O principal diferencial do AIfES (Artificial Intelligence for Embedded Systems) é sua capacidade de realizar tanto o treinamento quanto a inferência diretamente no dispositivo embarcado. Ele fornece todos os componentes necessários para treinar modelos de redes neurais, incluindo o algoritmo de retropropagação (*Backpropagation*) para as camadas, funções de perda (*loss functions*) e otimizadores (*optimizers*). Os autores validam essa capacidade ao treinar com sucesso redes neurais totalmente conectadas (FCNNs, *Fully Connected Neural Networks*) e redes neurais convolucionais (CNNs, *Convolutional Neural Networks*) em um MCU ARM Cortex-M4, demonstrando a viabilidade de treinar uma CNN com um consumo de RAM pouco superior a 100 kB.

3.3 Resumo Comparativo

Observa-se que cada ferramenta apresenta diferentes pontos fortes e fracos, dependendo do contexto de uso. O TFLM se destaca pela portabilidade e flexibilidade, o AIfES pela capacidade de treinamento embarcado, e o STM32Cube.AI pela otimização específica para MCUs STM32.

Em relação ao nível de suporte a diferentes MCUs, o único que não possui uma abordagem flexível é o STM32Cube.AI, que é exclusivo para a família STM32. Já o AIfES e o TFLM suportam uma ampla gama de arquiteturas, com o AIfES se destacando por sua arquitetura modular que facilita a adição de otimizações específicas para diferentes plataformas.

No quesito facilidade de uso os três trabalhos se diferenciam bastante, com o

[1] <https://github.com/Fraunhofer-IMS/AIfES_for_Arduino>

STM32Cube.AI sendo o mais acessível para iniciantes devido à sua interface gráfica e fluxo de trabalho simplificado. O TFLM, embora bem documentado, pode apresentar uma curva de aprendizado mais acentuada devido à sua complexidade técnica. O AIfES oferece um equilíbrio, com uma abordagem intuitiva para treinamento, mas um processo mais complexo para inferência.

No quesito otimização, todos trabalhos estão bem servidos , com o STM32Cube.AI se destacando por sua abordagem de geração de código que elimina o *overhead* do interpretador, enquanto o TFLM e o AIfES oferecem técnicas eficazes como quantização, alocação de memória estática e uso de bibliotecas otimizadas.

Por fim, apenas o AIfES oferece suporte ao treinamento diretamente no dispositivo embarcado, o que pode ser uma vantagem significativa para aplicações que exigem adaptação em tempo real ou aprendizado contínuo.

A Tabela 1 abaixo resume as principais características dos trabalhos analisados, utilizando os critérios definidos nessa seção.

Tabela 1 – Comparativo entre trabalhos

Trabalhos	**SUP**	**USO**	**OPT**	**TRE**
TFLM	Alto	Médio	Alto	Não
AIfES	Alto	Baixo	Alto	Sim
STM32Cube.AI	Baixo	Alto	Alto	Não
Este Trabalho	Alto	Alto	Alto	Não

O trabalho proposto por esta pesquisa busca combinar os pontos fortes do TFLM com a proposta da RL, criando uma forma mais acessível, otimizada, flexível e intuitiva de implementar modelos de ML em MCUs.

Em relação ao suporte a diferentes MCUs, este trabalho pode ser considerado de alto nível, por se basear no TFLM (que já oferece suporte a diversas arquiteturas) e na RL, cuja arquitetura é projetada para abstrair as especificidades do *hardware*.

Atualmente, a extensão desenvolvida possui suporte para MCUs da família STM32, mas foi validada apenas em ambiente de desenvolvimento, ou seja, arquitetura `x86_64`. Porém, a expectativa é que ela possa ser facilmente configurada para outras arquiteturas suportadas tanto pelo TFLM quanto pela RL. Isso ocorre porque o *wrapper* desenvolvido é independente do *hardware*, sendo o único fator limitante o suporte da RL às diferentes plataformas. Esse suporte, por sua vez, é constantemente ampliado com o desenvolvimento contínuo da RL.

No quesito facilidade de uso, a proposta do trabalho foi desde o início criar uma sintaxe simples e intuitiva, para facilitar a adoção por desenvolvedores com diferentes

níveis de experiência, portanto pode ser considerado de alto nível. Por fim, os recursos de otimização utilizados são os mesmos do TFLM, que são bastante eficazes, portanto também podem ser considerados de alto nível.

4 Implementação

Neste capítulo são apresentados os detalhes da implementação das funcionalidades propostas neste trabalho, incluindo a construção da camada de interoperabilidade, a implementação da biblioteca padrão em RL e as adaptações realizadas no *frontend* do compilador Robcmp.

O código-fonte completo pode ser encontrado no repositório online <https://github.com/LuizEduardoRezende/robcmp/tree/tflm-front-end>. O referido repositório é um *fork* do projeto Robcmp original, e todo o desenvolvimento pode ser encontrado na *branch* `tflm-front-end`. Futuramente, será submetido um *pull request* para integrar estas contribuições à *branch* principal do projeto Robcmp, a fim de que as novas funcionalidades fiquem disponíveis para a comunidade.

4.1 Arquitetura da Camada de Interoperabilidade

O processo de compilação e linkedição do Robcmp pode ser explicado da seguinte forma: programas escritos com a extensão `.rob` são compilados pelo Robcmp e geram arquivos objeto (`.o`). Já as bibliotecas externas, como o TFLM ou outras, são compiladas separadamente, resultando em arquivos de biblioteca estática (`.a`). Após a compilação de todos os programas e bibliotecas necessários, o *linker* entra em ação com o objetivo de resolver todas as referências entre os componentes e criar um executável único, que pode ser chamado de *firmware*.

É válido ressaltar que a biblioteca estática do TFLM pode ser facilmente compilada para diferentes arquiteturas, utilizando o sistema de `Makefile` do proprio *framework*. Diferente do arquivo objeto do programa em RL, que é gerado pelo próprio compilador Robcmp. Por fim, a ponte entre a RL e o TFLM é feita pelo *wrapper* C, que também é compilado por um sistema de Makefile dentro do diretório *test/tflm-tests*, gerando sua própria biblioteca estática.

A Figura 10 ilustra a arquitetura de compilação e interoperabilidade entre os componentes, as caixas azuis representam os códigos fontes dos diferentes componentes, enquanto que as caixas verdes representam os artefatos gerados após a compilação de cada componente. Após serem compilados, o linker unifica todos os artefatos em um único executável final.

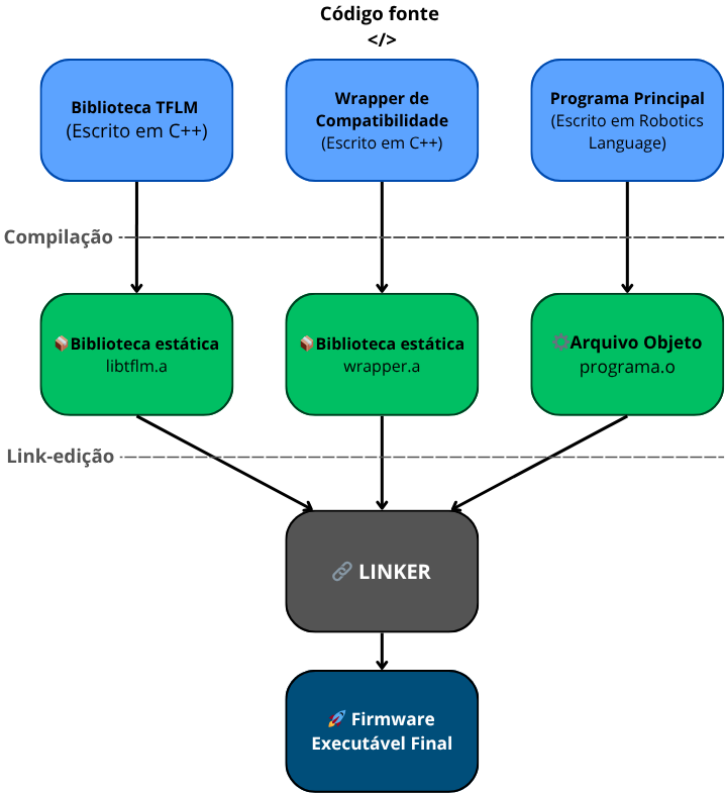

Figura 10 – Ilustração da arquitetura de compilação e interoperabilidade. As bibliotecas estáticas (`.a`) do TFLM e do *wrapper* C, assim como o arquivo objeto (`.o`) do programa em RL, são processados de forma independente e, na etapa final, unificados pelo *linker*. O *linker* resolve as referências entre os componentes para criar o executável único.

4.2 Implementação do *wrapper* C

O principal objetivo do *wrapper* C é fornecer uma interface simples e direta para que o código gerado pela RL possa interagir com a API do TFLM. Embora o arquivo-fonte tenha a extensão `.cpp` para permitir a chamada direta à API C++ do TFLM, todas as funções de interface foram declaradas com `extern "C"` para garantir uma ligação C (*C linkage*) pura. Essa abordagem permite que o código compilado da RL se conecte ao *wrapper* como se ele fosse uma biblioteca C nativa, resolvendo a interoperabilidade em tempo de linkagem. O código do *wrapper* pode ser encontrado no repositório do Robcmp, na pasta `/wrappers` ou no link direto <https://github.com/LuizEduardoRezende/robcmp/blob/tflm-front-end/wrappers/tflm/tflm_wrapper.cpp>.

4.2.1 Estrutura `TFLM_Instance`

A estrutura `TFLM_Instance` encapsula todos os objetos principais do TFLM, o interpretador, o alocador de memória e o resolvedor de operações. Isso mantém o estado de cada modelo carregado de forma organizada.

Código 4.1 – Struct `TFLM_Instance` encontrada no começo do *wrapper*.

```
1 struct TFLM_Instance {
2     tflite::MicroInterpreter* interpreter;
3     tflite::MicroAllocator* allocator;
4     MutableResolver* resolver;
5 };
```

4.2.2 Tipo enumerado `KernelType` e função `RegisterOp`

A enumeração `KernelType`, inspirada na enumeração `BuiltinOperator` do próprio TFLM, foi criada com o intuito de possibilitar o registro dinâmico de operadores (*kernels*) necessários para a execução de um modelo. Tanto a enumeração quanto a função `RegisterOp` atuam em conjunto para garantir que apenas os *kernels* essenciais, como `CONV_2D` ou `FULLY_CONNECTED`, sejam carregados dinamicamente conforme a necessidade especificada pelo desenvolvedor.

A função `RegisterOp` recebe como parâmetros o resolvedor de operações do TFLM e o tipo do *kernel* a ser registrado. Utilizando uma estrutura de controle `switch-case`, a função verifica o tipo do *kernel* e chama o método apropriado do resolvedor para registrá-lo. Além disso, ela contém diretivas de compilação condicional para incluir ou excluir o registro de determinados *kernels* com base em flags de compilação, como `ENABLE_ADD` ou `ENABLE_AVERAGE_POOL_2D`. Caso o *kernel* solicitado não esteja habilitado ou não seja suportado, a função imprime uma mensagem de aviso.

Código 4.2 – Tipo enumerado `KernelType` e função `RegisterOp`.

```
1  typedef enum {
2    ADD = 0,
3    AVERAGE_POOL_2D = 1,
4    CONCATENATION = 2,
5    CONV_2D = 3,
6    DEPTHWISE_CONV_2D = 4,
7    /* ... demais operadores ... */
8  } KernelType;
9
10 void RegisterOp(tflite::MicroMutableOpResolver<TFLM_MAX_OPS>* resolver, KernelType
       kernel_type) {
11     switch (kernel_type) {
12
13 #if ENABLE_ADD
14         case ADD:
```

```
15              resolver->AddAdd();
16              MicroPrintf("ADD registrado");
17              break;
18 #endif
19
20 #if ENABLE_AVERAGE_POOL_2D
21          case AVERAGE_POOL_2D:
22              resolver->AddAveragePool2D();
23              MicroPrintf("AVERAGE_POOL_2D registrado");
24              break;
25 #endif
26      /* ... demais casos ... */
27          default:
28              MicroPrintf("!!! Kernel %d nao suportado ou nao habilitado na
    compilacao", kernel_type);
29              break;
30      }
31 }
```

4.2.3 Configuração de Kernels: `kernel_config.h`

Para facilitar a configuração dos *kernels* disponíveis no *wrapper*, foi criado o arquivo de cabeçalho `kernel_config.h`. Nesse arquivo é possível habilitar ou desabilitar o suporte a cada *kernel* individualmente, através de diretivas de pré-processamento. Cada *kernel* possui uma diretiva `#define ENABLE_<NOME_DO_KERNEL>` conforme mostra o Código 4.3. Além disso, é possível habilitar ou desabilitar os logs de debug do TFLM, que são úteis para diagnóstico e depuração durante o desenvolvimento.

A diretiva `#pragma once` na primeira linha garante que o arquivo seja processado apenas uma vez pelo compilador evitando erros de duplicidade de definições.

Esse tipo de otimização é crucial em sistemas embarcados, pois permite reduzir o tamanho final do binário (*footprint*) ao excluir o código de operadores que não serão utilizados pelo modelo ou desabilitar os logs de debug. Isso garante que apenas os recursos estritamente necessários sejam linkados ao *firmware* final.

Código 4.3 – Arquivo de configuração `kernel_config.h`.

```
1  #pragma once
2
3  // Configuracao de logging
4  //1 = Habilitar logs de debug (MicroPrintf)
5  //0 = Desabilitar logs de debug
6  #define ENABLE_DEBUG_LOGS 0
7
8  // Configuracao de kernels
9  //1 = Habilitar kernel
10 //0 = Desabilitar kernel
11 #define ENABLE_ADD 0
12 #define ENABLE_AVERAGE_POOL_2D 1
13 #define ENABLE_CONCATENATION 0
14 #define ENABLE_CONV_2D 1
15 /* ... demais kernels ... */
```

4.2.4 `InitializeInterpreter`

É a primeira função a ser chamada em um fluxo de inferência, responsável por configurar o interpretador e outras instâncias necessárias para a execução do modelo, sua assinatura é apresentada no Código 4.4. Recebe como parâmetros os dados do modelo em formato de array de bytes, o buffer de memória pré-alocado (tensor arena) e a lista de *kernels* a serem registrados. Retorna um identificador único (*handle*) para a instância do modelo, do tipo `uintptr_t`, um inteiro sem sinal cujo tamanho varia de acordo com a arquitetura para a qual o *wrapper* foi compilado. Um inteiro é retornado porque o Robcmp não possui suporte a ponteiros nativos em sua linguagem.

É importante ressaltar que a função recebe alguns parâmetros adicionais além dos já mencionados, como `tensor_arena_size`, `num_kernels` e um parâmetro sem nome. Esses parâmetros podem ou não ser utilizados, mas devem constar na declaração da função por questões de compatibilidade com o *backend* do LLVM. As demais funções do *wrapper* também podem conter parâmetros extras pelo mesmo motivo.

A necessidade desses parâmetros surge quando o Robcmp utiliza a biblioteca padrão do diretório `lib/ai/tflm.rob`, que contém as declarações das funções do *wrapper* conforme mostrado na Seção 4.4. Essas declarações precisam estar alinhadas com as definições reais das funções no *wrapper*. No arquivo `.rob`, para cada argumento que seja um vetor, é adicionado um argumento extra do tipo inteiro que representa o tamanho desse vetor. Isso é uma característica do compilador Robcmp para tratar vetores em chamadas de função. Portanto, para manter a consistência entre declarações e definições, esses parâmetros adicionais são incluídos nas definições das funções no *wrapper*.

No caso de uso do *wrapper* sem a biblioteca padrão `tflm.rob`, ou seja, quando as chamadas são geradas diretamente pelo *frontend* do compilador, esses parâmetros adicionais não são necessários, pois as chamadas ocorrem diretamente no código C++ ou no LLVM-IR gerado pela análise semântica do compilador.

Código 4.4 – Assinatura da função `InitializeInterpreter`.

```
1 uintptr_t InitializeInterpreter(const uint8_t* model_data,
2                                 uint8_t* tensor_arena,
3                                 const uint8_t* required_kernels,
4                                 int,
5                                 int tensor_arena_size,
6                                 int8_t num_kernels);
```

4.2.5 `GetInputTensor` e `GetOutputTensor`

Estas são funções simples que retornam ponteiros para os tensores de entrada e saída do modelo, respectivamente. Esses ponteiros são utilizados para leitura e escrita dos dados posteriormente. As assinaturas das funções são mostradas no Código 4.5, elas recebem como parâmetro o *handle* do modelo e também o index do tensor desejado. Modelos podem ter mais de um tensor de entrada ou saída, ou seja , um modelo pode retornar diferentes arrays de saída dependendo da sua arquitetura.

Código 4.5 – Assinatura das funções `GetInputTensor` e `GetOutputTensor`.

```
1 uintptr_t GetOutputTensor(uintptr_t instance_handle, size_t index, int);
2
3 uintptr_t GetInputTensor(uintptr_t instance_handle, size_t index, int);
```

4.2.6 Convenção de Tipos de Dados e Quantização

Modelos de TinyML frequentemente usam tipos de dados otimizados, como inteiros de 8 bits, para economizar memória e acelerar o processamento. O *wrapper* simplifica isso para o usuário da RL, na medida em que permite que o desenvolvedor trabalhe sempre com valores de ponto flutuante (`float`), independentemente do tipo de dado real utilizado pelo modelo. Isso se mostra útil em situações onde os dados são recebidos de um sensor em ponto flutuante como temperatura, pressão ou aceleração.

As funções `SetTensorArray` e `SetTensorValue` do Código 4.7 convertem automaticamente valores de ponto flutuante para o formato quantizado exigido pelo modelo e vice-versa. O desenvolvedor em RL pode trabalhar sempre com float, e o *wrapper* cuida de toda a matemática de conversão internamente. Para tensores quantizados, o *wrapper* aplica o mapeamento linear baseado nos parâmetros de escala (`scale`) e ponto zero (`zero_point`) do tensor, conforme as fórmulas presentes no Código 4.6.

Apesar dessa estratégia de deixar o trabalho todo para o *wrapper* cobrir grande partes dos casos de uso, existem situações em que o modelo espera dados já quantizados, ou seja, dados já em inteiros de 8 bits. Nesses casos, a função `SetTensorFromIntArray` permite que o desenvolvedor forneça diretamente um array de inteiros, evitando a necessidade de conversão para ponto flutuante e posterior quantização. Isso é útil quando os dados já estão no formato quantizado, como em casos onde os dados são coletados de sensores que fornecem leituras em inteiros ou quando os dados são pré-processados externamente antes de serem passados ao modelo.

Para essas situações, a função `SetTensorFromIntArray` do Código 4.7 aceita um array de inteiros de 16 bits, que são então transformados para 8bits (apenas corta os bits

excedentes) e copiados diretamente para o tensor, sem qualquer conversão adicional.

Por fim, para o resgate dos valores após a inferência, as funções `GetTensorAsFloat` e `GetTensorArray` também realizam a desquantização automática(caso seja necessário), convertendo os valores quantizados de volta para ponto flutuante usando as mesmas fórmulas de mapeamento linear.

Código 4.6 – Formúlas de quantização e desquantização.

```
int32_t valor_quantizado = (int32_t)roundf((valor_float / scale) + (zero_point));

float valor_float = scale * (valor_quantizado - zero_point);
```

Código 4.7 – Assinatura das funções de resgate e passagem de valores.

```
void SetTensorValue(uintptr_t tensor_handle, size_t index, float value, int);

void SetTensorArray(uintptr_t tensor_handle, const float* values, size_t count, int);

void SetTensorFromIntArray(uintptr_t tensor_handle, const int16_t* values, size_t
    count, int);

float GetTensorAsFloat(uintptr_t tensor_handle, size_t index, int);

void GetTensorArray(uintptr_t tensor_handle, float* values, size_t max_count, int);
```

4.2.7 *InvokeInterpreter*

Essa é uma função simples responsável pela inferência do modelo. Ela aciona o TFLM para executar o modelo com os dados de entrada fornecidos. É o "cérebro" da operação, funcionando como um botão de ação. Sua assinatura está no Código 4.8, ela recebe apenas o *handle* do modelo como parâmetro.

Código 4.8 – Assinatura da função `InvokeInterpreter`.

```
void InvokeInterpreter(uintptr_t instance_handle, int);
```

4.2.8 *DestroyInterpreter*

Essa é uma função simples e semelhante a anterior, é responsável por liberar os recursos alocados para a instância do modelo, evitando vazamentos de memória. Sua assinatura está no Código 4.9, ela recebe apenas o *handle* do modelo como parâmetro.

Código 4.9 – Assinatura da função `DestroyInterpreter`.

```
void DestroyInterpreter(uintptr_t instance_handle, int);
```

4.3 Fluxo de Trabalho de Inferência

Para realizar uma inferência com um modelo TinyML utilizando o *wrapper*, o fluxo de trabalho segue os seguintes passos principais, por meio das chamadas de função correspondentes:

1. Inicialização (`InitializeInterpreter`): Esta é a etapa inicial do processo, onde são passados os dados necessários para configurar o modelo e é recebido um *handle* que representa a instância do modelo carregado.

2. Acesso aos tensores (`GetInputTensor` e `GetOutputTensor`): Com o *handle* do modelo, é possível obter ponteiros para os tensores de entrada e saída, que são utilizados para leitura e escrita dos dados.

3. Fornecimento dos dados de entrada (`SetTensorArray`) e (`SetTensorFromIntArray`): A partir dos *handles* dos tensores, os dados de entrada são inicializados.

4. Execução da inferência (`InvokeInterpreter`): Esta função é acionada para que os dados de entrada possam ser processados pelo modelo, gerando os resultados no tensor de saída.

5. Obtenção dos resultados (`GetTensorAsFloat` e `GetTensorArray`): Após a inferência, essas funções são usadas para ler os resultados do tensor de saída. Elas retornam o dado em formato de ponto flutuante.

6. Liberação de recursos (`DestroyInterpreter`): Essa função pode ser chamada para liberar os recursos para a instância do modelo, evitando vazamentos de memória.

4.4 Biblioteca padrão `tflm.rob`

A biblioteca padrão `tflm.rob` foi criada para servir como um passo intermediário entre a adaptação do *frontend* e o desenvolvimento do *wrapper* C. Ela contém as declarações de todas as funções do *wrapper*, como mostrado no Código 4.10, permitindo que programas escritos em RL chamem essas funções diretamente, sem a necessidade de modificar o compilador.

Ao declarar as funções no arquivo `tflm.rob`, o compilador passa a reconhecer sua existência e possibilita chamadas a partir do código `.rob`. As implementações reais estão presentes na biblioteca estática do *wrapper* C, utilizada em tempo de execução. Na etapa de linkedição, o *linker* resolve as referências entre o código objeto do programa e a biblioteca estática do *wrapper*, produzindo o executável final.

Além de facilitar a validação das funções do *wrapper*, a biblioteca `tflm.rob` também funciona como uma forma de apresentar o trabalho desenvolvido até o momento, servindo como mecanismo de segurança caso a modificação do *frontend* não esteja completa ou apresente problemas, antes da integração definitiva ao compilador.

Para utilizar alguma biblioteca padrão da RL, basta incluir no início do código a importação correspondente com a palavra chave `use` seguida do caminho relativo a pasta `lib`. No caso do TFLM, a linha de importação é `use "ai.tflm";`, uma vez que o arquivo se encontra na diretório `lib/ai/tflm.rob` do Robcmp.

Código 4.10 – Biblioteca padrão `tflm.rob`

```
enum KERNELS{
  ADD = 0,
  AVERAGE_POOL_2D = 1,
  CONCATENATION = 2,
  CONV_2D = 3,
  // demais operadores...
}

// Inicializacao com kernels especificos
uint64 InitializeInterpreter(uint8[] model_data, uint8[] tensor_arena, uint8[]
    required_kernels);

// Destruicao e limpeza
void DestroyInterpreter(uint64 instance_handle);

// Acesso aos tensores
uint64 GetInputTensor(uint64 instance_handle, uint64 index);
uint64 GetOutputTensor(uint64 instance_handle, uint64 index);

// Manipulacao de dados de entrada dos tensores
void SetTensorValue(uint64 tensor_handle, uint64 index, float value);
void SetTensorArray(uint64 tensor_handle, float[] values, uint64 count);
void SetTensorFromIntArray(uint64 tensor_handle, int16[] values, uint64 count);

// Manipulacao de dados de saida dos tensores
float GetTensorAsFloat(uint64 tensor_handle, uint64 index);
uint64 GetTensorSize(uint64 tensor_handle);
void GetTensorArray(uint64 tensor_handle, float[] values, uint64 max_count);

// Informacoes sobre tensores
uint64 GetInputTensorCount(uint64 instance_handle);
uint64 GetOutputTensorCount(uint64 instance_handle);

// Execucao de inferencia
void InvokeInterpreter(uint64 instance_handle);

// Funcoes de Analise de modelo
void DiagnoseModel(uint8[] model_data);
void VerifyModelData(uint8[] model_data);

// Funcao simples para imprimir float com quebras de linha
void PrintFloat(float value);
```

4.5 Ampliação da Robotics Language

As novas palavras chaves da RL foram criadas com o intuito de possibilitar uma experiência de desenvolvimento mais fluida, fácil e intuitiva para o usuário final. Com essas novas palavras chaves, o desenvolvedor consegue declarar um modelo, fornecer os dados de entrada, executar a inferência e obter os resultados de forma direta e simplificada.

A implementação da nova sintaxe foi realizada em caráter experimental para demonstrar sua viabilidade técnica. Entretanto, devido a limitações de tempo, os testes finais basearam-se na utilização da abordagem via biblioteca padrão.

O Código 4.11 mostra um exemplo simples de utilização das novas palavras-chave. Esse exemplo baseia-se em um uso real de um modelo TinyML disponível no repositório do TFLM: um modelo de predição de valores de seno denominado “hello_world_model”. Neste exemplo, primeiro são declaradas as constantes necessárias, como o tamanho da arena de memória e os *kernels* utilizados pelo modelo. Em seguida, o modelo é carregado a partir do arquivo TFLite, e os dados de entrada são fornecidos diretamente ao modelo. Após isso, a inferência é executada com a palavra-chave `invoke` e, finalmente, os resultados são obtidos diretamente do modelo.

Código 4.11 – Código proposto para a nova sintaxe que realiza a inferência de um modelo TFLite.

```
int16 main() {
  // 1. Define os dados de entrada, um buffer para a saida e o tamanho da arena.
  const arena_size = 8000;
  const kernels = {9};
  dados = {1.57};
  output = {0.0};

  // 2. Carrega o modelo
  model meu_modelo("meu_modelo.tflite", arena_size, kernels);

  // 3. Fornece a entrada para o modelo.
  meu_modelo.input[0] = dados;

  // 4. Executar a inferencia.
  invoke meu_modelo;

  // 5. Obtem o resultado.
  output = meu_modelo.output[0];
  //Resto do codigo
}
```

Para a implementação dessa nova sintaxe, foram necessárias modificações no *frontend* do compilador Robcmp, especificamente nas fases de análise léxica, sintática e semântica. Todo código fonte do *frontend* está localizado na pasta `src/`, portanto, todos os arquivos mencionados nesta seção podem ser encontrados nesse diretório do repositório do Robcmp.

4.5.1 Análise léxica

O arquivo Flex responsável pela análise léxica do Robcmp é o `Language.l`, cujo objetivo é identificar os tokens da linguagem RL. Foram adicionados quatro novos tokens para suportar as palavras-chave e as funcionalidades: `TOK_MODEL`, `TOK_INVOKE`, `TOK_MODEL_INPUT` e `TOK_MODEL_OUTPUT`.

Enquanto as expressões regulares para os tokens `TOK_MODEL` e `TOK_INVOKE` são simples, correspondendo diretamente às palavras-chave "model" e "invoke", as expressões para `TOK_MODEL_INPUT` e `TOK_MODEL_OUTPUT` são mais complexas: identificam padrões formados por um identificador seguido de ".input" ou ".output" e capturam o lexema completo para uso posterior nas análises sintática e semântica.

Código 4.12 – Novos *tokens* do *Scanner* Flex.

```
"model"          { return TOK_MODEL; }  // TFLM
"invoke"         { return TOK_INVOKE; } // TFLM

{ID}\.input      { yylval->ident = strndup(yytext, yyleng);
                 return TOK_MODEL_INPUT;
               }

{ID}\.output       { yylval->ident = strndup(yytext, yyleng);
                 return TOK_MODEL_OUTPUT;
               }
```

4.5.2 Análise sintática

No Robcmp, os arquivos responsáveis pela análise sintática são `Language.y`, `LanguageHeader.y` e `LanguageUse.y`. O primeiro define a gramática principal da linguagem RL, enquanto os dois últimos tratam, respectivamente, das declarações de cabeçalho e das regras sintáticas para a inclusão da biblioteca padrão.

Este trabalho concentrou-se principalmente no arquivo `Language.y`, onde foram adicionadas novas regras sintáticas à gramática principal da linguagem, e também no arquivo `LanguageHeader.y`, para suportar os novos tokens e novos tipos.

Inicialmente, antes de criar as novas regras sintáticas, foi necessário adicionar os novos tokens ao arquivo `LanguageHeader.y`, conforme mostrado no Código 4.13. Além disso, foi preciso definir um novo tipo de dado para representar os modelos TinyML, esse tipo recebeu o nome de `ModelNode`. Esse tipo foi adicionado ao arquivo `Header.y`, responsável por definir todos os tipos possíveis da AST de um programa, conforme mostrado no Código 4.14.

Código 4.13 – Novos tokens adicionados no `LanguageHeader.y`.

```
%token TOK_MODEL TOK_INVOKE TOK_MODEL_INPUT TOK_MODEL_OUTPUT
```

Código 4.14 – Novo tipo de dado definido no arquivo `Header.h`.

```
#include "ModelNode.h"
```

Após a fase de preparação, as novas regras sintáticas são definidas, juntamente com os novos tokens e seus respectivos tipos de dados no arquivo `Language.y`. Conforme mostrado no código Código 4.15, primeiramente os tokens são declarados e seus tipos de dado são definidos. Em seguida, as novas regras são implementadas na posição apropriada da gramática. A regra de declaração de um modelo (`model_stmt`) é adicionada às produções `global` e `stmt`, permitindo que modelos sejam declarados tanto no escopo global quanto dentro de funções ou blocos. Por outro lado, a regra de invocação de um modelo (`invoke_stmt`) é adicionada apenas à produção `stmt`, já que invocações de modelos não fazem sentido no escopo global. Além disso, as regras para manipulação de tensores são incorporadas nas produções `factor` e `complexvar_set`, permitindo que tensores de entrada e saída sejam tratados como variáveis complexas e como expressões, respectivamente.

Depois que o compilador reconhece uma sequencia de tokens que se encaixa nessas novas regras sintáticas, ele cria os nós correspondentes na AST. Esses nós são instâncias da classe `ModelNode`, que encapsula todas as informações relevantes sobre o modelo TinyML, como o nome do modelo, o tipo de operação (entrada, saída ou invocação), os parâmetros associados e a localização no código-fonte para fins de depuração.

Código 4.15 – Declaração dos tokens e das regras sintáticas no `Language.y`.

```
%type <node> model_stmt invoke_stmt
%type <ident> TOK_MODEL_INPUT TOK_MODEL_OUTPUT

global : use
       | function
       /* ... demais casos ... */
       | model_stmt ';'

stmt :
     /* ... demais casos ... */
     | model_stmt ';'
     | invoke_stmt ';'

factor :
       /* ... demais casos ... */
       | TOK_MODEL_OUTPUT[id] '[' expr ']' {
         // Extract model name from "model.output"
         std::string fullName($id);
         std::string modelName = fullName.substr(0, fullName.find('.'));
         $$ = new ModelNode(modelName.c_str(), "output", $expr, nullptr, @id);
         $$->setLocation(@id);
       }

model_stmt : TOK_MODEL TOK_IDENTIFIER '(' paramscall ')'  {
    $$ = new ModelNode($2, $4, @2);
 }
```

```
27
28 invoke_stmt : TOK_INVOKE TOK_IDENTIFIER  {
29     $$ = new ModelNode($2, "invoke", @2);
30  }
31
32 complexvar_set : TOK_MODEL_INPUT[id] '[' expr ']' '=' logicexpr {
33     // Extract model name from "model.input"
34     std::string fullName($id);
35     std::string modelName = fullName.substr(0, fullName.find('.'));
36     $$ = new ModelNode(modelName.c_str(), "input", $expr, $logicexpr, @id);
37     $$->setLocation(@id);
38 }
```

4.5.3 Análise semântica

A análise semântica no Robcmp é realizada por diversos arquivos C++ que são responsáveis por verificar as regras semânticas para diferentes tipos de nós na AST. Para criar as regras semânticas relacionadas aos modelos TinyML, foram criados os arquivos `ModelNode.h` e `ModelNode.cpp`, que definem a classe `ModelNode`. Essa classe herda da classe base `Node` e implementa os métodos necessários para a verificação semântica, geração de código LLVM-IR e outras funcionalidades.

O método mais importante dentro das classes/tipos de nós do Robcmp é o `generate`. Ele funciona como um visitor e é chamado pelo compilador ao percorrer a AST. No caso do `ModelNode`, dependendo das características do nó, o `generate` gera o código LLVM-IR correspondente para a declaração do modelo, invocação do modelo ou manipulação dos tensores de entrada e saída.

O comportamento da análise semântica se divide em duas operações principais: o processamento da declaração de um modelo (`model_stmt`) e o processamento do acesso aos seus membros (`.input`, `.output` e `invoke`). O método `generate` inicial é visto no Código 4.16, onde é decidido qual operação será realizada com base na existência dos `params`, que são os parâmetros: o nome do arquivo do modelo, o `tensor_arena` e a lista de *kernels*. A presença desses parâmetros indica uma declaração de modelo, enquanto a ausência deles indica um acesso aos membros, que também inclui a invocação do modelo.

Código 4.16 – Método `generate` da classe `ModelNode`.

```
1 Value* ModelNode::generate(FunctionImpl *func, BasicBlock *block, BasicBlock
      *allocblock) {
2     if (!getScope())  {
3         setScope(func);
4      }
5
6     if (params)  {
7         return generateDeclaration(func, block, allocblock);
8      }  else if (!memberName.empty())  {
9         return generateMemberAccess(func, block, allocblock);
10     }
```

```
11     return nullptr;
12 }
```

4.5.3.1 Processamento da Declaração do Modelo (`model_stmt`)

Esse método é responsável por gerar o código LLVM-IR necessário para declarar e inicializar uma instância do modelo TinyML. Ele realiza as seguintes etapas principais:

1. Validação do Arquivo de Modelo: o compilador verifica se o arquivo do modelo especificado nos parâmetros existe e é acessível. Caso contrário, um erro semântico é gerado.

2. Validação dos outros Parâmetros: o compilador verifica se os parâmetros `arena_size` e `kernels` existem e são válidos. Caso contrário, um erro semântico é gerado.

3. Geração de código LLVM-IR: cria várias variáveis globais que armazenam os dados do modelo, tamanho dos dados do modelo, `tensor_arena`, instância do interpretador e o array de *kernels*.

4. Chamada da função do *wrapper* C `InitializeInterpreter`: o compilador gera uma chamada para a função `InitializeInterpreter` do *wrapper* C, passando os parâmetros necessários, e armazena o *handle* retornado.

5. Retorna o *handle* do modelo: o método retorna o *handle* do modelo, que será utilizado em chamadas subsequentes para manipular o modelo.

4.5.3.2 Processamento de Acesso a Membros (`.input, .output, .invoke`)

Esse método é responsável por rotear o acesso aos membros dependendo do nome do membro acessado. Inicialmente o método verifica se o modelo foi declarado anteriormente, caso contrário, um erro semântico é gerado. Em seguida, dependendo do nome do membro acessado, o método chama uma das três funções específicas: `generateInputAccess`, `generateOutputAccess` ou `generateInvoke`. Caso o nome do membro não seja reconhecido, um erro semântico é gerado, conforme mostrado no Código 4.17.

Código 4.17 – Roteamento do método `generateMemberAccess`.

```
1     if (memberName == "input")  {
2         return generateInputAccess(func, block, allocblock, modelInstance);
3      }  else if (memberName == "output")  {
4         return generateOutputAccess(func, block, allocblock, modelInstance);
5      }  else if (memberName == "invoke")  {
6         return generateInvoke(func, block, allocblock, modelInstance);
```

```
7       } else {
8           yyerrorcpp("Membro '" + memberName + "' nao suportado no modelo '" +
    modelName + "'. Use 'input', 'output' ou 'invoke'.", this);
9           setSemanticError();
10          return nullptr;
11      }
```

4.5.3.3 *generateInputAccess (.input)*

Este método é responsável por realizar verificações quanto ao índice fornecido e à natureza do valor atribuído ao tensor de entrada. De acordo com as verificações, decide qual método de cópia de dados deve ser chamado. O método realiza as seguintes etapas principais:

1. Verifica se existe um valor à direita da atribuição: caso não exista o *assignedValue*, um erro semântico é gerado. Afinal, apenas atribuições são permitidas para o tensor de entrada.

2. Prepara o índice do tensor: caso o índice não seja fornecido, o valor padrão é 0.

3. Chama a função `GetInputTensor` e `GetTensorSize` do *wrapper* C: o compilador gera chamadas para essas funções, passando o *handle* do modelo e o índice do tensor, para obter o ponteiro para o tensor de entrada e seu tamanho.

4. Decide entre copiar um array ou um valor único: se o *assignedValue* for um array, é gerada uma chamada para o método `generateVariableArrayToTensorCopy`. Caso contrário, gera uma chamada para o método `generateScalarToTensorCopy`. Ambos métodos são responsáveis por gerar o código LLVM-IR para copiar os dados do valor atribuído para o tensor de entrada.

5. Tratamento de erros: em várias etapas o método detecta falhas de geração do código (índice, dados, etc), registra o erro semântico e retorna `nullptr`.

4.5.3.4 *generateOutputAccess (.output)*

Este método é responsável por gerar o código LLVM-IR responsável por resgatar valores do tensor de saída do modelo. Ele se assemelha bastante ao método `generateInputAccess`, com algumas diferenças importantes. O método realiza as seguintes etapas principais:

1. Prepara o índice do tensor: caso o índice não seja fornecido, o valor padrão é 0.

2. Chama a função `GetOutputTensor` do *wrapper* C: o compilador gera chamadas para essa função, passando o *handle* do modelo e o índice do tensor, para obter o ponteiro para o tensor de saída.

3. Chama a função `AllocAndGetTensorArray` do *wrapper* C: essa função recebe o ponteiro do tensor de saída, quantidade de elementos a serem resgatados (que pode ser nulo) e retorna um ponteiro para um array alocado dinamicamente contendo os valores em ponto flutuante.

4. Retorna o ponteiro do array: o método retorna o ponteiro para o array alocado dinamicamente.

4.5.3.5 `generateInvoke (invoke)`

Este método é responsável por gerar o código LLVM-IR necessário para invocar a inferência do modelo. Ele realiza as seguintes etapas principais:

1. Verifica se existe uma atribuição de valor: caso exista um *assignedValue*, um erro semântico é gerado. Afinal, a invocação do modelo não deve ter um valor atribuído.

2. Chama a função `InvokeInterpreter` do *wrapper* C: o compilador gera uma chamada para essa função, passando apenas o *handle* do modelo.

3. Retorna `nullptr`: como a invocação do modelo não produz um valor diretamente, o método retorna `nullptr`.

4.6 Considerações Finais

4.6.1 Limitações

Apesar da implementação ser funcional, algumas partes apresentam certas limitações e desvantagens que poderiam ter sido aprimoradas. No entanto, devido ao tempo disponível e ao grande escopo do projeto, algumas decisões foram tomadas visando a conclusão do trabalho.

Um aspecto que pode ser melhorado é a convenção adotada para a entrada e saída de dados nos modelos. Atualmente, caso o desenvolvedor utilize variáveis ou vetores de ponto flutuante, a responsabilidade pela conversão e quantização recai sobre o *wrapper*. Essa abordagem simplifica o uso da solução ao encapsular a complexidade, porém pode introduzir certas limitações. Por exemplo, ao utilizar métodos de arredondamento durante

a quantização ou conversão de tipos, pode ocorrer perda de precisão, o que deve ser considerado em aplicações sensíveis a erros.

Outra limitação está relacionada aos tokens que tratam o acesso aos tensores de entrada e saída de um modelo. Eles exigem uma sintaxe específica: um identificador seguido de ".input" ou ".output". Isso restringe a linguagem RL, impedindo que campos chamados `input` e `output` sejam declarados em tipos definidos pelo usuário (*types*, na terminologia da RL). O ideal seria não criar esses tokens na linguagem, mas sim tratar o acesso aos tensores como membros normais de um tipo definidos pelo usuário. Dessa forma, durante a análise semântica, o compilador verificaria se o identificador é um modelo, e então permitiria o acesso aos membros `input` e `output` normalmente.

Por fim, a classe `ModelNode` concentra uma grande quantidade de responsabilidades e utiliza um esquema de roteamento interno para decidir qual operação realizar. Isso resulta em um código menos modular e mais acoplado. O ideal seria dividir cada funcionalidade em classes separadas, como `ModelDeclarationNode`, `ModelInvokeNode`, `ModelInputNode` e `ModelOutputNode`.

Algumas dessas limitações podem ser melhoradas em trabalhos futuros. Isso inclui um refatoramento do fluxo do *frontend* e a criação de novas estratégias no *wrapper* para contornar a perda de precisão durante a quantização.

4.6.2 Portabilidade e Suporte a Diferentes Arquiteturas

Este trabalho tem como objetivo inicial implementar o suporte para o TFLM na arquitetura ARM Cortex-M, utilizada por diversas famílias de microcontroladores, como a STM32. No entanto, o desenvolvedor pode escolher a arquitetura desejada para gerar o *firmware*, desde que o Robcmp e o TFLM ofereçam suporte à plataforma escolhida. Para isso, é necessário compilar tanto o *wrapper* quanto a biblioteca do TFLM para a arquitetura específica, conforme demonstrado na Seção 5.2. Dessa forma, o sistema pode ser adaptado para diferentes plataformas, bastando seguir o procedimento de compilação adequado.

5 Avaliação e Testes

Este capítulo abordará tudo relacionado ao processo de avaliação e testes da solução proposta. Inicialmente, serão descritos os procedimentos metodológicos adotados, incluindo também quais foram os recursos utilizados na execução dos testes, como *hardware*, *software*, modelos de ML e outras ferramentas relevantes. Em seguida, será detalhada a configuração do ambiente experimental, fornecendo informações sobre como replicar o ambiente utilizado para os testes. Por fim, serão apresentados os resultados obtidos durante os testes, acompanhados de uma análise crítica e uma discussão final sobre os resultados.

5.1 Metodologia

Esta seção descreve a metodologia adotada para avaliação e testes da solução, apresentando o ambiente de desenvolvimento (*software*, *hardware* e modelos), bem como os procedimentos de validação funcional e a análise comparativa de eficiência.

5.1.1 Software e Hardware Utilizados

Os novos recursos e funcionalidades implementados diretamente na linguagem RL foram desenvolvidos com o auxílio das ferramentas Flex (versão 2.6.4) e Bison (versão 3.8.2), responsáveis por gerar os analisadores léxico e sintático do Robcmp. Além disso, a análise semântica foi escrita em C++ conforme a estrutura do compilador. Por outro lado, a nova biblioteca padrão foi desenvolvida com a própria linguagem RL, criando um arquivo `.rob` que continha todas as declarações das funções do *wrapper* C do TFLM.

O ambiente de desenvolvimento foi o Visual Studio Code (versão 1.106.1), com o auxílio de duas extensões principais: o PlatformIO (versão 6.1.18), para depurar e exportar os *firmwares* para as placas, e o RobCmpSyntax (versão 1.0), que fornece o realce de sintaxe para arquivos .rob. Além disso, para a validação dos experimentos, foram utilizados *scripts* de automação (Makefiles personalizados) que padronizaram a execução dos testes unitários, garantindo que as flags de otimização (como `-Oz`) e linkedição fossem aplicadas consistentemente em todas as amostras comparativas.

A máquina de desenvolvimento utilizada foi um *notebook* Lenovo IdeaPad 3 15ITL6, equipado com processador Intel® Core™ i7-1165G7, 16 GB de RAM e sistema operacional Ubuntu 24.04.1 LTS. As validações funcionais e de eficiência foram realizadas nesta máquina, gerando binários compatíveis com a arquitetura `x86`.

5.1.2 Modelos de Machine Learning Adotados

Para garantir a abrangência dos testes, foram selecionados diferentes modelos de TinyML, variando em complexidade e tipo de dado de entrada (numérico, texto, áudio e imagem). Alguns foram selecionados do repositório do próprio TFLM no Github, enquanto que outros foram adquiridos na internet. Os modelos utilizados são:

Preditor de Seno: O modelo "Hello World" do TinyML. É um modelo simples que tem como objetivo prever valores de uma onda senoidal com base em valores de entrada x no intervalo de 0 a 2π.

Preditor de Seno Quantizado: Uma variação do modelo anterior, porém submetido ao processo de quantização pós-treinamento. Seus pesos e ativações foram convertidos de ponto flutuante para inteiros (8 bits).

Classificador de Spam: Esse classificador de spam é um modelo de Processamento de Linguagem Natural (NLP, do inglês *Natural Language Processing*) que categoriza mensagens de texto como "spam" ou "não spam". Baixado do site Kaggle, é um modelo demonstrativo gerado pelo TensorFlow Lite Model Maker, treinado a partir de um arquivo de vocabulário contendo a lista de palavras e seus tokens equivalentes. Ele recebe como entrada um tensor de 20 inteiros (Int32) representando tokens de uma sentença codificada. Sua saída é um vetor de ponto flutuante (Float32) contendo duas probabilidades: a de ser spam e a de não ser.

Micro Speech: O Micro Speech é CNN compacta (< 20 kB) projetada para reconhecer as palavras-chave "yes" e "no". Ele opera em conjunto com um modelo pré-processador que converte amostras de áudio bruto (16 KHz) em espectrogramas compostos por 49 features. O classificador recebe esses dados como entrada e retorna as probabilidades para quatro categorias: silence, unknown, yes e no.

Person Detection: Este é um modelo de detecção de pessoas que utiliza uma rede neural de aproximadamente 250 kB para reconhecer a presença humana. Ele recebe como entrada um tensor de inteiros de 8 bits (int8) contendo os dados brutos de uma imagem em escala de cinza. O modelo processa esse vetor e gera como saída pontuações (scores) que classificam a cena entre as categorias "pessoa" e "não pessoa".

5.1.3 Procedimentos de Validação Funcional

A etapa de validação funcional teve como objetivo verificar se a solução proposta cumpria o que prometia, sem levar em consideração sua eficiência, aplicabilidade, facilidade ou qualquer outro fator que mede sua qualidade.

Nesta etapa, as novas funcionalidades foram validadas utilizando testes unitários, que foram executados na máquina de desenvolvimento. Esses testes foram projetados para verificar se cada nova funcionalidade implementada no Robcmp estava operando conforme o esperado. Foram criados diversos arquivos `.rob` que utilizavam diferentes modelos de ML, abrangendo uma variedade de cenários e casos de uso. Os testes foram desenvolvidos utilizando apenas a biblioteca padrão ai.tflm disponibilizada pelo Robcmp, uma vez que as adaptações no *frontend* do compilador não estavam totalmente integradas para a realização dos testes finais.

5.1.4 Validação da eficiência e Análise Comparativa

Para validar a eficiência da integração do TFLM à RL, foi realizada uma análise comparativa entre duas abordagens de desenvolvimento de *firmwares*: os desenvolvidos com a biblioteca padrão do Robcmp para TinyML e os desenvolvidos diretamente em C++ utilizando o ambiente do TFLM. Nesta avaliação, foram considerados critérios como a facilidade de desenvolvimento, a manutenibilidade do código, o tempo de execução e o tamanho final do executável.

Para calcular o tempo de execução foi utilizada a ferramenta de *benchmark* de linha de comando Hyperfine. Diferente do utilitário padrão *time*, o Hyperfine automatiza a execução múltipla do binário para garantir relevância estatística, calculando a média e o desvio padrão dos tempos de execução. A ferramenta foi configurada para executar uma fase de aquecimento(*warmup*) de 10 iterações antes da medição, visando mitigar a latência de I/O (leitura de disco) e estabilizar o uso de cache do processador. Em seguida foram realizadas 100 iterações de medição com o binário em questão para obter uma média confiável do tempo de execução.

Para atingir resultados verídicos, os binários foram executados isoladamente, sem outros processos concorrentes na máquina de desenvolvimento. O tempo de medição utilizado foi a média do tempo real de execução (*Time (mean)*) fornecido pelo Hyperfine.

5.2 Configuração do Ambiente Experimental

Para facilitar o processo de compilação e execução dos testes, foi estabelecido um ambiente de desenvolvimento padronizado. Um guia detalhado com o passo a passo para a instalação das ferramentas, compilação do compilador modificado e configuração das dependências encontra-se disponível no Apêndice B.

A estrutura do projeto foi organizada de modo que os testes relacionados ao TFLM ficassem isolados no diretório `test/tflm-tests`. Neste diretório, foi alocado um `Makefile`

personalizado, responsável por automatizar todo o processo de compilação, linkedição e execução dos testes.

Este `Makefile` executa as seguintes etapas principais para cada um dos arquivos de teste `.rob` presentes no diretório:

1. Compila o código do *wrapper* C do TFLM para gerar a biblioteca estática somente uma vez (caso ela não exista ou alguma configuração do *wrapper* tenha sido modificada);
2. Invoca o compilador Robcmp para traduzir os arquivos fonte `.rob` em código objeto (*.o*);
3. Realiza a linkedição do código objeto gerado com as bibliotecas estáticas do TFLM (`libtensorflow-microlite.a`) e do *wrapper* de compatibilidade desenvolvido.
4. Executa todos os binários gerados, exibindo *PASS* caso o retorno seja 0, ou *FAILED* caso seja diferente de 0.

Para garantir a otimização do tamanho do binário final e a compatibilidade com o TFLM em ambiente embarcado, foram utilizadas diversas *flags* de compilação e linkedição. As principais opções adotadas e suas respectivas funções estão descritas a seguir:

`-fno-rtti` e `-fno-exceptions`: Desabilitam informações de tipo em tempo de execução e o suporte a exceções do C++ na compilação do *wrapper*.

`-DTF_LITE_STATIC_MEMORY=1`: Define uma macro que instrui o TFLM a utilizar alocação de memória estática, o que é preferível em sistemas embarcados e necessário para compatibilidade com a RL.

`-ffunction-sections` e `-fdata-sections`: Forçam o compilador a colocar cada função e item de dados em sua própria seção de memória.

`-Oz`: Flag de compilação dos arquivos `.rob`. Nível de otimização agressivo focado na redução do tamanho do código gerado.

`-Wl,-gc-sections`: Realiza a coleta de lixo (*garbage collection*) de seções não utilizadas.

`-Wl,-strip-debug`: Remove informações de depuração do binário final.

`-Wl,-discard-all`: Descarta todas as seções não essenciais do binário final.

`-Wl,-build-id=none`: Remove o identificador de build do binário final.

`-Wl,-relax` : Permite ao *linker* otimizar instruções e endereços para reduzir o tamanho do código final.

`-flto=thin`: Habilita a Otimização em Tempo de Linkedição (*Link Time Optimization*), permitindo que o otimizador analise o programa como um todo, e não apenas arquivo por arquivo.

O ambiente foi configurado para gerar executáveis compatíveis com a arquitetura `x86_64`, permitindo a validação apenas na máquina de desenvolvimento.

5.3 Resultados da Validação Funcional

Todos os testes unitários relacionados ao TFLM estão localizados na pasta `test/tflm-tests`. Essa separação foi necessária pois o processo de linkedição dos programas que utilizam o TFLM é diferente dos programas convencionais, exigindo a inclusão da biblioteca estática do TFLM e do *wrapper* C.

Dentro do diretório de testes desejado, basta executar o comando `make` para que todos os testes sejam compilados e executados automaticamente. Como convenção, os arquivos de teste que utilizam a nova sintaxe possuem o prefixo `syntax-`, enquanto os que utilizam a biblioteca padrão possuem o prefixo `lib-`.

Um arquivo de exemplo foi colocado no apêndice no final deste trabalho que realiza os testes unitários com o modelo de classificação de spam. O Código A.1 apresenta o arquivo `lib-spam.rob`, que utiliza a biblioteca padrão, todos os outros testes unitários com a biblioteca padrão seguem a mesma lógica.

Esses testes verificam se o modelo de ML é carregado corretamente, se as entradas são processadas adequadamente e se as saídas correspondem ao esperado, entre outros aspectos. Todos os testes unitários com a biblioteca padrão foram executados com sucesso, exibindo a mensagem *PASS* ao final de cada execução, confirmando que as novas funcionalidades implementadas estão operando conforme o esperado.

5.4 Resultados da Validação da Eficiência e Análise Comparativa

A Tabela 2 apresenta o tamanho do binário final gerado para cada teste unitário, enquanto a Tabela 3 exibe o tempo de execução médio obtido em cada um. Ambos os testes foram realizados na máquina de desenvolvimento.

Com base nos resultados apresentados, observa-se que a utilização da biblioteca padrão em RL introduz um *overhead* natural devido à camada de abstração e ao *wrapper* de compatibilidade.

Em relação ao tamanho do binário, o aumento absoluto variou entre 5 KB e 16 KB. No modelo *Person Detection*, o maior testado, a diferença de 5 KB representou um aumento de apenas 1,15% (439 KB contra 434 KB), demonstrando que mesmo para aplicações reais de maior porte, o impacto no armazenamento é negligenciável.

Quanto ao tempo de execução, nota-se um comportamento constante em todos os modelos testados, a diferença absoluta variou minimamente entre 0,914 ms e 1,414 ms, o que pode ser considerado relevante em aplicações de tempo real ou com restrições rigorosas de latência. No entanto, é importante destacar que, para modelos mais complexos como o *Person Detection*, essa diferença representa apenas cerca de 0,94% do tempo total de execução (150,674 ms contra 149,260 ms), indicando que o impacto relativo diminui conforme a complexidade do modelo aumenta.

Tabela 2 – Tamanho do binário (*bytes*) na máquina de desenvolvimento

Modelo TinyML	**Biblioteca padrão**	**TFLM(C++)**
Preditor de Seno quantizado	94K	88K
Preditor de Seno	94K	88K
Classificador de Spam	199K	188K
Micro Speech	149K	133K
Person Detection	439K	434K

Tabela 3 – Tempo de execução (ms) na máquina de desenvolvimento

Modelo TinyML	**Biblioteca padrão**	**TFLM(C++)**
Preditor de Seno quantizado	1.921	1.007
Preditor de Seno	1.916	0.985
Classificador de Spam	1.973	1.058
Micro Speech	10.914	9.851
Person Detection -	150.674	149.260

No quesito facilidade de desenvolvimento e manutenibilidade do código, a utilização da biblioteca padrão em RL demonstrou ser significativamente mais vantajosa. Uma comparação pode ser feita entre o Código A.1 (`.rob` com biblioteca padrão) e Código A.2 (C++ com TFLM) apresentados nos apêndices do trabalho.

No desenvolvimento em C++, o programador precisa ter conhecimento detalhado da arquitetura do *framework* para que consiga manipular corretamente os símbolos e funções do TFLM. Para que seja desenvolvida uma aplicação funcional é necessário: incluir múltiplos arquivos de cabeçalho específicos, instanciar manualmente o registrador de operadores e registrar operador por operador, gerenciar explicitamente o ciclo de vida do

intérprete e a alocação de tensores, realizar verificações de versão de schema (TFLITE_-SCHEMA_VERSION), acessar ponteiros brutos e iterar sobre eles para manipular os dados de entrada e saída, entre outras tarefas complexas.

Em contrapartida, o desenvolvimento utilizando a biblioteca padrão em RL é significativamente mais simples e direto. Com uma única linha de importação da biblioteca `ai.tflm`, o programador tem acesso a todas as funcionalidades necessárias para criar uma aplicação TinyML. O registro de operadores (*kernels*) é feita de forma declarativa através de um array de IDs, eliminando a necessidade de chamadas de função repetitivas. Além disso, na inicialização do modelo, o processo de configuração do intérprete e alocação de tensores é abstraído, permitindo que o desenvolvedor foque apenas na lógica da aplicação. A manipulação dos dados de entrada e saída é realizada de maneira intuitiva, utilizando arrays nativos da RL, sem a necessidade de lidar com ponteiros brutos ou estruturas complexas.

Apesar do código em RL possuir mais caracteres por conta da ausência de um sistema de arquivos para incluir os dados do modelo e dados de entrada diretamente (como é feito em C++), o número de linhas de código é significativamente menor, o que facilita a leitura e manutenção do código.

5.5 Considerações Finais

A integração do TFLM à linguagem RL, por meio da biblioteca padrão desenvolvida, foi validada com sucesso na máquina de desenvolvimento através de testes funcionais e uma análise comparativa de eficiência. Os resultados indicam que a solução proposta se mostra funcional e eficaz, proporcionando uma alternativa viável para o desenvolvimento de aplicações TinyML em RL.

Apesar do aumento no tamanho do binário e no tempo de execução, os impactos foram considerados aceitáveis, especialmente quando comparados aos benefícios em termos de facilidade de desenvolvimento e manutenibilidade do código.

O plano inicial era realizar os testes tanto na máquina de desenvolvimento quanto em uma placa embarcada com o microcontrolador STM32F407VET6. No entanto, devido a limitações de tempo, os testes em ambiente embarcado não foram concluídos. Recomenda-se que futuros trabalhos explorem essa etapa, avaliando o desempenho e a eficiência da solução em um ambiente real de aplicação TinyML.

6 Conclusões e Trabalhos Futuros

Este trabalho realizou a extensão da RL através da integração da biblioteca TFLM. Essa extensão ocorreu de duas formas distintas: uma concluída e a outra parcialmente implementada. A primeira foi a criação da biblioteca padrão `ai.tflm` para o Robcmp; já a segunda foi a proposta e o planejamento da adaptação do *frontend* do compilador para suportar novas palavras-chave e tipos de dados, permitindo a escrita de programas em RL que utilizem o TFLM de forma nativa.

O desenvolvimento da biblioteca `ai.tflm` foi bem-sucedido. Ela utiliza a camada de interoperabilidade explicada na Seção 4.1 e o *wrapper* desenvolvido na Seção 4.2 para expor as funcionalidades do TFLM ao Robcmp. A validação ocorreu na máquina de desenvolvimento por meio de testes unitários, garantindo funcionalidade e confiabilidade para diferentes modelos de TinyML. Esses modelos processam diversos tipos de entrada, como números simples, texto tokenizado e a representação em arrays de bytes de imagens e espectrogramas de áudio. Além disso, a eficiência foi avaliada em termos de tamanho do binário e tempo de execução, demonstrando a adequação da biblioteca para dispositivos com recursos limitados.

A adaptação do *frontend* do Robcmp para suportar o TFLM nativamente não foi concluída. No entanto, grande parte já foi implementada e detalhada durante o texto. A implementação dessa funcionalidade permitiria aos desenvolvedores escrever programas em RL que utilizem o TFLM de forma mais intuitiva, sem a necessidade de recorrer à biblioteca `ai.tflm`. Essa abordagem nativa facilitaria a integração de modelos TinyML em aplicações RL, promovendo uma experiência de desenvolvimento mais fluida e eficiente.

Pode-se concluir que o desenvolvimento do *wrapper* e da biblioteca `ai.tflm` representa uma base sólida para o desenvolvimento de aplicações TinyML em RL. Isso permite que aplicações de ML sejam desenvolvidas para dispositivos embarcados com recursos limitados, aproveitando as capacidades do TFLM e também da RL que inclui desenvolvimento de *firmware* com baixo acoplamento, alta coesão, elevada manutenibilidade e abstração de *hardware*.

Como possíveis melhorias a serem desenvolvidas para o compilador Robcmp e sua linguagem RL, tanto no contexto de TinyML quanto em outras áreas, destacam-se:

- Finalização da implementação do *frontend* do Robcmp para suportar o TFLM nativamente, conforme planejado.
- Validação da biblioteca `ai.tflm` e do *frontend* adaptado em dispositivos embarcados reais, avaliando seu desempenho e eficiência em ambientes com recursos limitados.

- Exploração de outras bibliotecas de ML para integração com o Robcmp, ampliando as opções disponíveis para desenvolvedores.
- Criação de novas estratégias no *wrapper* desenvolvido com o intuito de: modernizar a otimização de tamanho de binário e contornar a perda de precisão durante a quantização e conversão de tipos.

Com base nas contribuições deste trabalho, espera-se que o Robcmp e a linguagem RL possam continuar a evoluir, incorporando novas funcionalidades e melhorias que atendam às necessidades dos desenvolvedores de sistemas embarcados e aplicações de TinyML. A integração do TFLM é um passo importante nessa direção, e futuras pesquisas podem explorar ainda mais as possibilidades em áreas como TinyML, *Edge AI* e sistemas IoT inteligentes.

Referências

ABADADE, Y. et al. A comprehensive survey on tinyml. *IEEE Access*, IEEE, v. 11, p. 96892–96922, 2023. Citado 2 vezes nas páginas 16 e 17.

AHO, A. V. et al. *Compiladores: princípios, técnicas e ferramentas*. 2. ed. São Paulo: Pearson Addison-Wesley, 2008. 648 p. ISBN 978-85-88639-24-9. Citado 2 vezes nas páginas 25 e 26.

ATTARAN, N. et al. Embedded low-power processor for personalized stress detection. *IEEE Transactions on Circuits and Systems II: Express Briefs*, v. 65, n. 12, p. 2032–2036, 2018. Citado 2 vezes nas páginas 15 e 30.

AXELSON, J. *The Microcontroller Idea Book: Circuits, Programs & Applications Featuring the 8052-BASIC Microcontroller*. [S.l.]: lakeview research llc, 1997. Citado na página 15.

BIGLARI, A.; TANG, W. A review of embedded machine learning based on hardware, application, and sensing scheme. *Sensors*, v. 23, n. 4, 2023. ISSN 1424-8220. Disponível em: <https://www.mdpi.com/1424-8220/23/4/2131>. Citado na página 15.

Cambridge University Press. *Cambridge Dictionary: Mainstream*. 2025. <https://dictionary.cambridge.org/dictionary/english/mainstream>. Acesso em: 19 mai. 2025. Citado na página 30.

COOPER, K. *Construindo Compiladores*. GEN LTC, 2013. ISBN 9788535255645. Disponível em: <https://books.google.com.br/books?id=M2cIrgEACAAJ>. Citado 8 vezes nas páginas 9, 20, 21, 22, 23, 24, 25 e 27.

CORPORATION, A. *Company History*. 2025. <https://www.microchip.com/en-us/about/corporate-overview/acquisitions/atmel>. Acessado em: 24 set. 2025. Citado na página 29.

DAVID, R. et al. Tensorflow lite micro: Embedded machine learning for tinyml systems. In: SMOLA, A.; DIMAKIS, A.; STOICA, I. (Ed.). *Proceedings of Machine Learning and Systems*. [s.n.], 2021. v. 3, p. 800–811. Disponível em: <https://proceedings.mlsys.org/paper_files/paper/2021/file/6c44dc73014d66ba49b28d483a8f8b0d-Paper.pdf>. Citado 3 vezes nas páginas 17, 32 e 37.

FISCHER, C.; CYTRON, R.; LEBLANC, R. *Crafting a Compiler*. Addison-Wesley, 2010. (Crafting a compiler with C). ISBN 9780136067054. Disponível em: <https://books.google.com.br/books?id=G4Y_AQAAIAAJ>. Citado 3 vezes nas páginas 9, 21 e 22.

GAUTHIER, T.; KALISZYK, C.; URBAN, J. Proving theorems with neural networks as guidance. *Nature*, v. 607, n. 7917, p. 687–692, 2022. Citado na página 30.

GIUSTO, D. et al. (Ed.). *The Internet of Things, 20th Tyrrhenian Workshop on Digital Communications*. Berlin: Springer, 2010. ISBN 978-1-4419-1673-0. Citado na página 28.

HOANG, T. M.; NAM, S. H.; PARK, K. R. Enhanced detection and recognition of road markings based on adaptive region of interest and deep learning. *IEEE Access*, v. 7, p. 109817–109832, 2019. Citado na página 15.

HUSSAIN, A. et al. Programming a microcontroller. *Int. J. Comput. Appl*, v. 155, n. 5, p. 21–26, 2016. Citado 2 vezes nas páginas 15 e 28.

International Organization for Standardization (ISO); International Electrotechnical Commission (IEC); Institute of Electrical and Electronics Engineers (IEEE). Norma Internacional, *ISO/IEC/IEEE 24765: Systems and software engineering — Vocabulary*. 2017. Citado na página 33.

MARWEDEL, P. *Embedded System Design: Embedded Systems Foundations of Cyber-Physical Systems, and the Internet of Things*. 4th. ed. Springer, 2021. (Embedded Systems). 1st edition: Springer US 2006; 2nd edition: Springer Netherlands 2011; 3rd edition: Springer International Publishing 2018. ISBN 978-3-030-60909-2. Disponível em: <https://doi.org/10.1007/978-3-030-60910-8>. Citado 2 vezes nas páginas 27 e 28.

Microchip Technology Inc. *AVR® 8-bit MCUs*. 2025. <https://www.microchip.com/en-us/products/microcontrollers-and-microprocessors/8-bit-mcus/avr-mcus>. Acessado em: 25 de maio de 2025. Citado 2 vezes nas páginas 9 e 29.

OLIVEIRA, T. Borges de. *RobCMP - Robotic Compiler*. 2018. <https://github.com/thborges/robcmp/>. Acesso em: 14 abril de 2025. Disponível em: <https://github.com/thborges/robcmp/>. Citado 2 vezes nas páginas 18 e 34.

RODRIGUES, D. M. *Elaboração de uma Linguagem de Programação Específica para Robótica Educacional*. 65 p. Monografia — Universidade Federal de Goiás, Regional Jataí, Jataí, GO, Brasil, 2018. Citado na página 34.

RUSSELL, S. J.; NORVIG, P. *Artificial Intelligence: A Modern Approach*. Pearson, 2020. (Pearson series in artificial intelligence). ISBN 9780134610993. Disponível em: <https://books.google.com.bn/books?id=koFptAEACAAJ>. Citado na página 30.

SANTOS, R. F. M. dos. *Design and Implementation of the Standard Library for a Domain-Specific Programming Language in Robotics: Mathematical Functions*. 49 p. Monografia — Universidade Federal de Jataí, Jataí, GO, Brasil, 2024. Citado 2 vezes nas páginas 34 e 36.

SINGH, R.; GILL, S. S. Edge ai: A survey. *Internet of Things and Cyber-Physical Systems*, v. 3, p. 71–92, 2023. ISSN 2667-3452. Disponível em: <https://www.sciencedirect.com/science/article/pii/S2667345223000196>. Citado 2 vezes nas páginas 16 e 30.

SORO, S. Tinyml for ubiquitous edge ai. *arXiv preprint arXiv:2102.01255*, 2021. Citado 4 vezes nas páginas 15, 16, 17 e 32.

SOUSA, W. G. de. *Avaliação Técnica de Componentes Eletrônicos e Microprocessadores para uso no Processo de Ensino-Aprendizagem de Ciências Exatas*. 55 p. Monografia — Universidade Federal de Goiás, Regional Jataí, Jataí, GO, Brasil, 2018. Citado na página 34.

STMicroelectronics. *UM2526 - Getting started with X-CUBE-AI Expansion Package for Artificial Intelligence (AI)*. [S.l.], 2021. User manual, Rev 7. Disponível em: <https://www.st.com/resource/en/user_manual/um2526-getting-started-with-xcubeai-expansion-package-for-artificial-intelligence-ai-stmicroelectronics.pdf>. Citado na página 38.

STMicroelectronics. *STM32F103x8 STM32F103xB: Medium-density performance line Arm®-based 32-bit MCU with 64 or 128 KB Flash, USB, CAN, 7 timers, 2 ADCs, 9 com. interfaces*. [S.l.], 2023. Production data. Citado 2 vezes nas páginas 9 e 29.

STMicroelectronics. *Arm Microcontrollers - 32-bit MCUs - STMicroelectronics*. 2025. <https://www.st.com/content/st_com/en/arm-32-bit-microcontrollers.html>. Acessado em: 25 de maio de 2025. Última atualização informada pela página: 2025. Citado na página 29.

STOCKFISH. *Stockfish: Open Source Chess Engine*. 2025. <https://stockfishchess.org/>. Acesso em: 24 set. 2025. Disponível em: <https://stockfishchess.org/>. Citado na página 30.

SUBHI, V. A. *Majestic: Uma Ampliação de uma Linguagem de Programação para Robótica Educacional*. 57 p. Monografia — Universidade Federal de Goiás, Regional Jataí, Jataí, GO, Brasil, 2019. Citado 2 vezes nas páginas 34 e 36.

Tesla, Inc. *Tesla Autopilot AI*. 2025. <https://www.tesla.com/AI>. Acesso em: 24 set. 2025. Disponível em: <https://www.tesla.com/AI>. Citado na página 30.

The LLVM Project. *LLVM Language Reference Manual*. 2025. <https://llvm.org/docs/LangRef.html>. Acessado em 18 de maio 2025. Citado 3 vezes nas páginas 9, 26 e 27.

VIEIRA, G. K. de A. *Design and Implementation of the Standard Library for a Domain-Specific Programming Language in Robotics: String Manipulation*. 57 p. Monografia — Universidade Federal de Jataí, Jataí, GO, Brasil, 2024. Citado 2 vezes nas páginas 34 e 36.

WARDEN, P.; SITUNAYAKE, D. *TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-Low-Power Microcontrollers*. Sebastopol: O'Reilly Media, 2019. Disponível em: <https://www.oreilly.com/library/view/tinyml/9781492052036/>. Citado 2 vezes nas páginas 16 e 32.

WU, Z.; QIU, K.; ZHANG, J. A smart microcontroller architecture for the internet of things. *Sensors*, v. 20, n. 7, 2020. ISSN 1424-8220. Disponível em: <https://www.mdpi.com/1424-8220/20/7/1821>. Citado na página 15.

WULFERT, L. et al. Aifes: A next-generation edge ai framework. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, v. 46, n. 6, p. 4519–4533, 2024. Citado na página 39.

Apêndices

APÊNDICE A – Testes Unitários

Código A.1 – Teste unitário do modelo classificador de spam com a biblioteca padrão `tflm.rob`

```
use ai.tflm;

spam_model_data = {0x1c,0x0,0x0,0x0,0x54
// (Conteudo do modelo em bytes omitido para brevidade)
};

spam_model_data_size = 83304;

// Arena de memoria para tensores (16KB para garantir espaco suficiente)
tensor_arena = {0u:4096};

// Kernels necessarios para o modelo Spam, Gather, Mean, FullyConnected e Softmax
required_kernels = {36, 40, 9, 25};


int8 test_init() {
    h = InitializeInterpreter(spam_model_data, tensor_arena, required_kernels);
    if(h == 0){
        return 1;
    }
    DestroyInterpreter(h);
    return 0;
}

// Testa inferencia para frase nao SPAM
int8 test_infer_not_spam() {
    h = InitializeInterpreter(spam_model_data, tensor_arena, required_kernels);
    if(h == 0) {
        return 1;
    }

    in = GetInputTensor(h, 0);
    out = GetOutputTensor(h, 0);

    if(in == 0 or out == 0) {
        DestroyInterpreter(h);
        return 2;
    }
    // "This is a great article thank you for the information"
    dados = {1.0, 82.0, 70.0, 214.0, 55.0, 98.0, 84.0, 47.0, 89.0, 26.0, 73.0, 0.0,
    0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0};
    output = {0.0, 0.0};

    SetTensorArray(in, dados, 20);
    InvokeInterpreter(h);
    GetTensorArray(out, output, 2);
    DestroyInterpreter(h);

    PrintFloat(output[1]);

    if (output[1] > 0.1) {
        return 3;
    }
    return 0;
}

// Testa inferencia para frase SPAM
int8 test_infer_spam() {
    h = InitializeInterpreter(spam_model_data, tensor_arena, required_kernels);
    if(h == 0) {
```

```
60          return 1;
61      }
62
63      in = GetInputTensor(h, 0);
64      out = GetOutputTensor(h, 0);
65
66      if(in == 0 or out == 0) {
67          DestroyInterpreter(h);
68          return 2;
69      }
70      // Frase: "win free money online click www site com"
71      dados = {1.0, 93.0, 326.0, 72.0, 8.0, 485.0, 24.0, 35.0, 6.0, 0.0, 0.0, 0.0, 0.0,
        0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0};
72      output = {0.0, 0.0};
73
74      SetTensorArray(in, dados, 20);
75      InvokeInterpreter(h);
76      GetTensorArray(out, output, 2);
77      DestroyInterpreter(h);
78
79      PrintFloat(output[1]);
80
81      if(output[1] < 0.90) {
82          return 3;
83      }
84      return 0;
85  }
86
87  // Testa falha ao obter tensor de entrada (indice invalido)
88  int8 test_input_tensor_fail() {
89      h = InitializeInterpreter(spam_model_data, tensor_arena, required_kernels);
90      if(h == 0) {
91          return 1;
92      }
93      in = GetInputTensor(h, 99); // indice invalido
94      DestroyInterpreter(h);
95      if(in != 0) {
96          return 2;
97      }
98      return 0;
99  }
100
101 // Testa falha ao obter tensor de saida (indice invalido)
102 int8 test_output_tensor_fail() {
103     h = InitializeInterpreter(spam_model_data, tensor_arena, required_kernels);
104     if(h == 0) {
105         return 1;
106     }
107     out = GetOutputTensor(h, 99); // indice invalido
108     DestroyInterpreter(h);
109     if(out != 0) {
110         return 2;
111     }
112     return 0;
113 }
114
115 // Executa todos os testes unitarios
116 int8 main() {
117     result = 0;
118
119     if(test_init() != 0){ result = 1;}
120     if(test_infer_not_spam() != 0) {result = 2;}
121     if(test_infer_spam() != 0) {result = 3;}
122     if(test_input_tensor_fail() != 0) {result = 4;}
123     if(test_output_tensor_fail() != 0) {result = 5;}
124
125     PrintFloat(float(result));
126
127     return result; // Todos passaram
```

```
128 }
```

Código A.2 – Teste unitário do modelo classificador de spam em C++ com o TFLM

```
1 #include "tensorflow/lite/core/c/common.h"
2 #include
    "tensorflow/lite/micro/examples/firmware_spam_detector/models/spam_model_data.h"
3 #include "tensorflow/lite/micro/micro_interpreter.h"
4 #include "tensorflow/lite/micro/micro_log.h"
5 #include "tensorflow/lite/micro/micro_mutable_op_resolver.h"
6 #include "tensorflow/lite/micro/system_setup.h"
7 #include "tensorflow/lite/schema/schema_generated.h"
8
9 namespace {
10 using SpamDetectorOpResolver = tflite::MicroMutableOpResolver<4>;
11
12 TfLiteStatus RegisterOps(SpamDetectorOpResolver& op_resolver) {
13   TF_LITE_ENSURE_STATUS(op_resolver.AddGather());
14   TF_LITE_ENSURE_STATUS(op_resolver.AddMean());
15   TF_LITE_ENSURE_STATUS(op_resolver.AddFullyConnected());
16   TF_LITE_ENSURE_STATUS(op_resolver.AddSoftmax());
17   return kTfLiteOk;
18 }
19 }  // namespace
20
21 // Global variables for tests
22 constexpr int kTensorArenaSize = 4 * 1024;
23 uint8_t tensor_arena[kTensorArenaSize];
24 SpamDetectorOpResolver op_resolver;
25
26 // Test sentences
27 int32_t test_sentence_1[20] = {1, 82, 70, 214, 55, 98, 84, 47, 89, 26, 73, 0, 0, 0,
    0, 0, 0, 0, 0, 0}; // "This is a great article thank you for the information"
28 int32_t test_sentence_2[20] = {1, 93, 326, 72, 8, 485, 24, 35, 6, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0}; // "win free money online click www site com"
29
30 // Test initialization
31 int test_init() {
32   TF_LITE_ENSURE_STATUS(RegisterOps(op_resolver));
33
34   const tflite::Model* model = ::tflite::GetModel(g_spam_model_data);
35   if (model->version() != TFLITE_SCHEMA_VERSION) {
36     return 1;
37   }
38
39   tflite::MicroInterpreter interpreter(model, op_resolver, tensor_arena,
       kTensorArenaSize);
40   if (interpreter.AllocateTensors() != kTfLiteOk) {
41     return 1;
42   }
43
44   return 0;
45 }
46
47 // Test inference for non-spam sentence
48 int test_infer_non_spam() {
49   const tflite::Model* model = ::tflite::GetModel(g_spam_model_data);
50   if (model->version() != TFLITE_SCHEMA_VERSION) {
51     return 1;
52   }
53
54   tflite::MicroInterpreter interpreter(model, op_resolver, tensor_arena,
```

```
    kTensorArenaSize);
  if (interpreter.AllocateTensors() != kTfLiteOk) {
    return 1;
  }

  TfLiteTensor* input = interpreter.input(0);
  TfLiteTensor* output = interpreter.output(0);

  if (input == nullptr || output == nullptr) {
    return 2;
  }

  // Set non-spam sentence
  for (int j = 0; j < 20; ++j) {
    input->data.i32[j] = test_sentence_1[j];
  }

  if (interpreter.Invoke() != kTfLiteOk) {
    return 2;
  }

  float not_spam_prob = output->data.f[0];
  float spam_prob = output->data.f[1];

  MicroPrintf("test_infer_non_spam: not_spam_prob=%f, spam_prob=%f",
    (double)not_spam_prob, (double)spam_prob);

  // Expect non-spam probability to be higher
  if (spam_prob > not_spam_prob) {
    return 3;
  }

  return 0;
}

// Test inference for spam sentence
int test_infer_spam() {
  const tflite::Model* model = ::tflite::GetModel(g_spam_model_data);
  if (model->version() != TFLITE_SCHEMA_VERSION) {
    return 1;
  }

  tflite::MicroInterpreter interpreter(model, op_resolver, tensor_arena,
    kTensorArenaSize);
  if (interpreter.AllocateTensors() != kTfLiteOk) {
    return 1;
  }

  TfLiteTensor* input = interpreter.input(0);
  TfLiteTensor* output = interpreter.output(0);

  if (input == nullptr || output == nullptr) {
    return 2;
  }

  // Set spam sentence
  for (int j = 0; j < 20; ++j) {
    input->data.i32[j] = test_sentence_2[j];
  }

  if (interpreter.Invoke() != kTfLiteOk) {
    return 2;
  }
```

```
115
116   float not_spam_prob = output->data.f[0];
117   float spam_prob = output->data.f[1];
118
119   MicroPrintf("test_infer_spam: not_spam_prob=%f, spam_prob=%f",
        (double)not_spam_prob, (double)spam_prob);
120
121   // Expect spam probability to be higher
122   if (not_spam_prob > spam_prob) {
123     return 3;
124   }
125
126   return 0;
127 }
128
129 // Test failure when getting input tensor (invalid index)
130 int test_input_tensor_fail() {
131   const tflite::Model* model = ::tflite::GetModel(g_spam_model_data);
132   if (model->version() != TFLITE_SCHEMA_VERSION) {
133     return 1;
134   }
135
136   tflite::MicroInterpreter interpreter(model, op_resolver, tensor_arena,
        kTensorArenaSize);
137   if (interpreter.AllocateTensors() != kTfLiteOk) {
138     return 1;
139   }
140
141   TfLiteTensor* input = interpreter.input(99); // invalid index
142
143   if (input != nullptr) {
144     return 2;
145   }
146
147   return 0;
148 }
149
150 // Test failure when getting output tensor (invalid index)
151 int test_output_tensor_fail() {
152   const tflite::Model* model = ::tflite::GetModel(g_spam_model_data);
153   if (model->version() != TFLITE_SCHEMA_VERSION) {
154     return 1;
155   }
156
157   tflite::MicroInterpreter interpreter(model, op_resolver, tensor_arena,
        kTensorArenaSize);
158   if (interpreter.AllocateTensors() != kTfLiteOk) {
159     return 1;
160   }
161
162   TfLiteTensor* output = interpreter.output(99); // invalid index
163
164   if (output != nullptr) {
165     return 2;
166   }
167
168   return 0;
169 }
170
171 // Main function that runs all tests
172 int main(int argc, char* argv[]) {
173   tflite::InitializeTarget();
174
```

```
  int result = 0;

  if (test_init() != 0) { result = 1; }
  if (test_infer_non_spam() != 0) { result = 2; }
  if (test_infer_spam() != 0) { result = 3; }
  if (test_input_tensor_fail() != 0) { result = 4; }
  if (test_output_tensor_fail() != 0) { result = 5; }

  MicroPrintf("Test result: %d", result);
  if (result == 0) {
    MicroPrintf("All tests passed successfully!");
  }

  return result;
}
```

APÊNDICE B – Guia de Instalação e Configuração

Este apêndice detalha os passos necessários para configurar o ambiente de desenvolvimento, isto envolve compilar o fork do Robcmp e replicar o sistema de build utilizado nos experimentos.

No futuro, um novo desenvolvedor não precisará seguir estes passos pois o Robcmp já disponibilizará todas as possíveis bibliotecas pré-compiladas do *wrapper* e do TFLM. Assim, o processo de desenvolvimento poderá ser feito através de forma automática e integrada ao PlatformIO, da mesma forma que já ocorre para programas embarcados convencionais.

Porém, este apêndice ainda é valioso por ensinar o processo de instalação dos pacotes necessários, clonagem correta do repositório e compilação do Robcmp, tarefas obrigatórias para um novo desenvolvedor. Além disso, este conteúdo é útil para trabalhos futuros que desejem modificar o compilador Robcmp ou entender melhor sua integração com o TFLM.

B.1 Pré-requisitos de Software

Antes de iniciar, é necessário garantir que os seguintes pacotes de *software* estejam instalados no sistema (preferencialmente um ambiente Linux, como o Ubuntu):

- Git: Para controle de versão e download do repositório.
- CMake: Para configurar o sistema de build do Robcmp.
- Make: Para executar os scripts de compilação e linkedição.
- LLVM (versão 20+): Conjunto de ferramentas e bibliotecas para compilação.
- Clang (versão 20+): Compilador C/C++ baseado no LLVM.
- Flex (versão 2.6.4 ou similar): Gerador de analisador léxico.
- Bison (versão 3.8.2 ou similar): Gerador de analisador sintático.

A maioria desses pacotes pode ser instalada em sistemas baseados em Debian/Ubuntu com os comandos:

```
sudo apt-get update
sudo apt-get install git cmake make clang llvm flex bison
```

B.2 Download do Repositório e Submódulos

O fork do Robcmp utiliza submódulos do Git para gerenciar a dependência do TFLM. É necessário iniciar o submódulo existente após o clone com os seguintes comandos:

```
# Clona o repositorio fork do Robcmp
git clone https://github.com/LuizEduardoRezende/robcmp.git

# Navega ate o diretorio do repositorio
cd robcmp

# Seleciona a branch com o frontend do TFLM
git checkout tflm-front-end

# Inicializa e atualiza os submodulos
git submodule update --init --recursive
```

Talvez seja necessário atualizar(*refresh*) o seu explorador de arquivos para visualizar os novos arquivos criados, após a inicialização e atualização dos submódulos.

B.3 Processo de Compilação (*Build*)

O processo de compilação é dividido em duas etapas, conforme a arquitetura descrita na Seção 4.1. Primeiro, compila-se a biblioteca estática do TFLM; em seguida, compila-se o Robcmp.

Caso esteja buscando compilar para uma arquitetura especifica (ex: Cortex-M4), é possível verificar todas as arquiteturas suportadas pelo TFLM na documentação do *GitHub* [1].

Com uma arquitetura específica em mente, navegue até o diretório do submódulo do TFLM, depois use o sistema de Makefile do TFLM para compilar a biblioteca para a arquitetura desejada:

[1] <https://github.com/tensorflow/tflite-micro/blob/main/tensorflow/lite/micro/cortex_m_generic/README.md>

```
# Navega ate o diretorio do TFLM
cd third_party/tflite-micro/

# Compila para uma plataforma especifica
make -f tensorflow/lite/micro/tools/make/Makefile TARGET=cortex_m_generic
    TARGET_ARCH=cortex-m4 microlite

# Compila para a plataforma de sua maquina de desenvolvimento(linux,
    windows, etc)
make -f tensorflow/lite/micro/tools/make/Makefile microlite
```

Ao final deste processo, a biblioteca estática com nome padronizado deverá estar presente no diretório `third-party/tflite-micro/gen/(nome-da-arquitetura)/lib/libtensorflow-microlite.a`. Talvez seja necessário atualizar(*refresh*) o seu explorador de arquivos para visualizar os novos arquivos criados.

Atenção, caso tenha feito a compilação de mais de uma biblioteca estática do TFLM, escolha apenas uma arquitetura para trabalhar por vez. Para que o build e a configuração do ambiente de desenvolvimento ocorram corretamente é necessário que exista apenas uma arquitetura dentro de `third-party/tflite-micro/gen`. Caso deseje trabalhar com multiplas arquiteturas, é recomendado que se crie cópias do repositório do Robcmp para cada arquitetura desejada.

Com a biblioteca TFLM pronta, retorne ao diretório raiz do robcmp para compilar o projeto principal utilizando o CMake e o Make:

```
# Cria o diretorio de build e navega ate ele
mkdir build
cd build

# Configura o projeto
cmake ..

# Compila o projeto (use -j para compilacao paralela, ex: -j6)
make
```

Se a compilação for bem-sucedida, o executável do compilador (arquivo `robcmp`) estará disponível dentro do diretório `build`, pronto para ser usado nos testes de validação.

B.4 Compilação de arquivos .rob que usam o TFLM com Makefile

Para conseguir compilar um programa `.rob` criado ou já existente basta utilizar o `Makefile` localizado no diretório `test/tflm-tests`. Este Makefile consegue compilar programas que estejam localizados dentro dessa mesma pasta. A partir desse diretório utilize os seguintes comandos:

```
# Mostra todos os comandos disponiveis
make help

# Compila e executa todos os testes unitarios presentes na pasta
make

# Remove os arquivos objeto(.o) e executaveis gerados por programas
make clean

# Executa o programa que foi gerado na pasta out
./out/syntax-spam
```