Calebe Miranda Ferreira Braga de Castro

Refinamento e Integração do Modelo LLM Compiler na Otimização de Código Alvo para Microcontroladores AVR na Robotics Language

Jataí-GO

2025

Calebe Miranda Ferreira Braga de Castro

Refinamento e Integração do Modelo LLM Compiler na Otimização de Código Alvo para Microcontroladores AVR na Robotics Language

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.

Universidade Federal de Jataí

Orientador: Prof. Dr. Thiago Borges de Oliveira

Jataí-GO

2025

Dedico este trabalho a todos que me apoiaram e acreditaram em mim durante esta jornada. Ao meu irmão e à minha família, que sempre me motivaram e incentivaram com amor e paciência, durante todo o processo. E também, ao professor e orientador Thiago Borges de Oliveira, que me apoiou e forneceu inestimáveis instruções.

Agradecimentos

Gostaria de agradecer aos meus amigos e familiares que estiveram comigo durante todo o processo de realização deste trabalho, em especial ao estudante Alisson Ferreira Teles, que realizou um trabalho com tema semelhante e foi um grande parceiro durante todos os momentos difíceis desta caminhada. Agradeço à minha mãe, Lara Michelle Ferreira Braga, pelo amor incondicional e pelas palavras de motivação, assim como agradeço a todos os meus professores e avaliadores pela confiança para a realização deste trabalho. Muito Obrigado !

A ciência da computação não é sobre computadores. É sobre humanidade.

- Donald Knuth

Resumo

Nos últimos anos, avanços em modelos de linguagem de grande escala (LLMs) tornaram possível aplicar técnicas de inteligência artificial a tarefas antes exclusivas de mãos humanas, incluindo a geração e transformação de código. Trabalhos recentes, em especial o modelo LLM-Compiler, demonstraram que LLMs treinados sobre representações intermediárias de compiladores são capazes de sugerir sequências de passes de otimização e, em alguns casos, gerar código otimizado, tornando o processo de compilação mais inteligente. Entretanto, esses esforços foram majoritariamente dirigidos a arquiteturas de propósito geral, incompatíveis com microcontroladores usados em sistemas embarcados, que devido a restrições severas de recursos se beneficiariam muito desta nova tecnologia. Diante dessa lacuna, este trabalho propõe e implementa um pipeline reprodutível dedicado a preparar o material necessário para o refinamento supervisionado do LLM-Compiler para o domínio AVR (com foco no ATmega328P), permitindo adaptar a capacidade dos LLMs às limitações e particularidades de MCUs. O pipeline integra as etapas de geração e coleta de programas em C (códigos sintéticos e reais), conversão automática para a Robotics Language (ROBL) via software `c2rob`, compilação para LLVM-IR direcionada ao backend AVR, execução de uma busca controlada sobre um conjunto convertido de 1.289 sequências de otimização e geração automática de pares *prompt*/*label* no formato adequado para *Supervised Fine-Tuning*. Como artefatos, o trabalho entrega scripts automatizados, dados consolidados (pares JSONL e tabelas de melhores sequências) e documentação que viabilizam a execução posterior do fine-tuning em infraestruturas apropriadas. Em termos de contribuição, o principal resultado é a disponibilização de um ecossistema preparado para futuras pesquisas sobre otimização de código para microcontroladores. O conjunto de ferramentas e datasets produzidos permite avaliar e comparar estratégias de otimização específicas ao domínio AVR, além de servir como base para experimentos de fine-tuning que visem melhorar a geração de sequências de passes em sistemas embarcados.

Palavras-chaves: LLM; otimização de compiladores; LLVM-IR; microcontroladores AVR; pipeline; fine-tuning.

Abstract

In recent years, advances in large language models (LLMs) have made it possible to apply artificial intelligence techniques to tasks previously performed exclusively by humans, including code generation and transformation. Recent works, particularly the LLM-Compiler model, have demonstrated that LLMs trained on compiler intermediate representations are capable of suggesting optimization pass sequences and, in some cases, generating optimized code, making the compilation process more intelligent. However, these efforts have been directed primarily at general-purpose architectures, which are incompatible with the constraints of microcontrollers used in embedded systems—devices that, due to their severe resource limitations, would greatly benefit from this new technology. Addressing this gap, this work proposes and implements a reproducible pipeline dedicated to preparing all necessary material for the supervised refinement of the LLM-Compiler in the AVR domain (focused on the ATmega328P), enabling LLMs to be adapted to the limitations and particularities of MCUs. The pipeline integrates the stages of generating and collecting C programs (both synthetic and real), automatic conversion to the Robotics Language (ROBL) via `c2rob` software, compilation into LLVM-IR targeting the AVR backend, execution of a controlled search over a converted set of 1,289 optimization sequences, and the automatic creation of *prompt/label* pairs in a format suitable for *Supervised Fine-Tuning*. As artifacts, this work delivers automated scripts, consolidated data (JSONL pairs and tables of best sequences), and documentation that enables future fine-tuning on appropriate computational infrastructures. In terms of contribution, the main result is the availability of an ecosystem designed to support future research on code optimization for microcontrollers. The tools and datasets produced allow researchers to evaluate and compare optimization strategies tailored to the AVR domain, while also serving as a foundation for fine-tuning experiments aimed at improving optimization pass generation in embedded systems.

Keywords: LLM; compiler optimization; LLVM-IR; AVR microcontrollers; pipeline; fine-tuning.

Lista de ilustrações

Lista de abreviaturas e siglas

IA	Inteligência Artificial
LLM	*Large Language Models*/Grandes modelos de linguagem
MCU	*Microcontroller Unit*/Microcontroladores
CPU	*Central Processing Unit*
RAM	*Random Access Memory*
ROM	*Read-Only Memory*
ROBL	*The Robotics Language*
IoT	*Internet of Things*
LLVM	*Low Level Virtual Machine*
LLVM-IR	*Low Level Virtual Machine-Intermediate Representation*
YaCoS	*Yet another Compiler Optimization Search*

Sumário

1 Introdução

1.1 Motivação

Modelos de Inteligência Artificial (IA) têm sido amplamente utilizados em diversas tarefas no contexto da ciência da computação. Nos últimos anos, os avanços em Modelos de Linguagem de Grande Escala ou LLMs (*Large Language Models*) tornaram possível aplicar IA em atividades antes consideradas exclusivamente humanas, como geração de código, edição de projetos, sugestão e análise de codificação e, mais recentemente, otimização de código para compiladores.

Seria possível um modelo de linguagem treinado com exemplos de código de fases intermediárias do compilador substituir ou melhorar as estratégias de otimização empregadas por compiladores tradicionais? Cummins et al. (2023) partiram desta pergunta fundamental e treinaram um modelo baseado na arquitetura LLaMA 2[1] com 7 bilhões de parâmetros, alimentado por uma base de dados de funções LLVM-IR[2], ou seja, códigos de linguagem intermediária gerados normalmente após a análise semântica de um compilador, não otimizados, associados a boas sequências de passes de otimização obtidas por busca semi-exaustiva (*autotuning*), além dos respectivos códigos otimizados. O principal objetivo era avaliar a possibilidade de um modelo de IA gerar a sequência correta de passes de otimização que diminuiria o tamanho do código binário resultante. O modelo treinado, chamado LLM-Compiler, foi capaz de gerar listas de passes de otimização que superaram as obtidas por estratégias como -Oz do LLVM, com uma média de melhoria de 3% a 5% na contagem de instruções, sem a necessidade de realizar múltiplas compilações como ocorre nos métodos tradicionais de *autotuning*. Além disso, ao ser perguntado sobre qual seria o código otimizado resultante da aplicação dos passes de otimização, o modelo demonstrou capacidade de raciocínio sobre código, atingindo mais de 90% de sucesso na geração de código compilável e uma taxa de equivalência exata de 70% com o código gerado pelo compilador usando os mesmos passes.

Os resultados de Cummins et al. (2023) se mostram promissores, visto que a *flag* -Oz é uma *flag* de otimização utilizada em diversos compiladores atuais como, por exemplo, Clang (que usa o *backend* LLVM), com o objetivo específico de gerar o menor tamanho de código binário possível durante a compilação. Existem outras *flags* comumente utilizadas como -Os, -O1, -O2 e -O3, sendo o -Os uma *flag* de otimização que tenta reduzir o tamanho do código sem comprometer muito a performance, e as *flags* de otimização de execução -O1, -O2 e -O3, que otimizam para uma velocidade boa, intermediária e

[1] Site da Meta Llama 2: <https://www.llama.com/llama2/>
[2] Site do manual de referência da linguagem: <https://llvm.org/docs/LangRef.html>

rápida de execução de código, respectivamente, porém, também aumentam o tamanho do código binário produzido e tornam o processo de compilação proporcionalmente mais lento. Como mostrado por Wang e O'Boyle (2018), as *flags* de otimização tradicionais apresentam limitações. Os autores destacam que essas configurações genéricas muitas vezes não oferecem o melhor desempenho possível para todos os programas ou arquiteturas devido ao fato de aplicarem um conjunto fixo de transformações, sem considerar as características específicas do código-fonte ou do hardware de destino. Segundo Wang e O'Boyle (2018), técnicas de aprendizado de máquina permitirão a seleção e ordenação dinâmica de passes de otimização com base em características específicas do código fonte e do ambiente de execução, personalizando o processo de compilação e maximizando ainda mais o desempenho. Sendo assim, é possível perceber que essa tecnologia é promissora e útil.

Atualmente, grande parte dos equipamentos de tecnologia em nossas residências e empresas compartilham uma característica em comum: em alguma instância, fazem uso de sistemas embarcados. Presentes em automóveis, telefones, impressoras, sensores, equipamentos médicos e uma infinidade de outros dispositivos eletrônicos essenciais para a indústria, comunicação, pesquisa e funcionamento de uma nação, sistemas embarcados são pequenos sistemas computacionais desenvolvidos para realizar uma tarefa específica, desde ligar um ar condicionado até auxiliar o controle de um drone (LI; YAO, 2003). O cerne de um sistema embarcado são os microcontroladores ou MCUs (*Microcontroller Units*). Microcontroladores são circuitos integrados que compõem o "cérebro" de sistemas embarcados. Uma MCU possui uma CPU (*Central Processing Unit*), uma memória RAM (*Random Access Memory*), alguma forma de memória de longo prazo ou ROM (*Read-Only Memory*) e métodos de entrada e saída (HUSSAIN et al., 2016).

Segundo Li e Yao (2003), microcontroladores desempenham um papel fundamental no funcionamento da sociedade atual e o mundo de hoje não funcionaria sem os *softwares* embarcados em nossos aparelhos. Logo, qualquer processo de otimização no funcionamento de MCUs e, consequentemente, de sistemas embarcados, resultaria em um enorme ganho para toda a sociedade. Porém, devido à sua natureza de atuação específica, as MCUs são extremamente restritas em relação ao poder de processamento e armazenamento de dados e necessitam de soluções eficientes de otimização de código (WANG; O'BOYLE, 2018).

O LLM-Compiler, desenvolvido por (CUMMINS et al., 2023), embora seja uma solução promissora para atender os requisitos de otimização de código em geral, foi projetado e treinado no domínio de sistemas computacionais comuns, ou seja, para otimizar código para CPUs convencionais de 64 bits, X86_64 e AArch64. Tais CPUs possuem um nível de capacidade computacional elevado comparado com microcontroladores. Por exemplo, a CPU Intel Core i9-10900K que implementa uma arquitetura de 64 bits, com diversas extensões ao conjunto de instruções, como Intel SSE4.1, Intel SSE4.2 e Intel AVX2, possui

uma memória cache de 20 MB e suporta memória RAM DDR4 de até 128 GB, com uma frequência base de *clock* de 3.70 GHz (Intel,). Em comparação, arquiteturas como a AVR RISC de 8 bits, foco de atuação deste trabalho, presente em microcontroladores como o ATmega328P, possuem características extremamente limitadas, como 32 KB de memória *flash* programável, 2 KB de SRAM interna, e um *clock* máximo de 16 MHz (Atmel Corporation, 2015). Sendo assim, levando em consideração a utilidade de sistemas embarcados, que utilizam MCUs, e também os resultados otimistas produzidos pelo trabalho de Cummins et al. (2023), surge a necessidade de se adaptar o LLM-Compiler para suprir a demanda por otimização de *software* em microcontroladores de sistemas embarcados.

Considerando estas demandas, o projeto *Robotics Language:* Uma Linguagem de Programação de Propósito Específico para Microcontroladores (PI05974-2024), desenvolvido na Universidade Federal de Jataí, assim como seu antecessor, Especificação e Construção de Protótipos Funcionais de Kits Robóticos de Baixo Custo para uso em Processos de Ensino-Aprendizagem (PI02361-2018), tem como objetivo desenvolver uma linguagem de programação, chamada *The Robotics Language* (ROBL), e seu compilador, denominado Robcmp, específica para robótica e microcontroladores.

Para o exercício deste trabalho, a capacidade do Robcmp de abstrair aspectos específicos do *hardware* e permitir um desenvolvimento compatível com diversos microcontroladores de forma dinâmica e sua integração com o *backend* AVR do LLVM (OLIVEIRA, 2024), o torna a linguagem ideal para servir como ferramenta de conversão e compilação do conjunto de códigos de entrada e das respostas na saída do modelo.

1.2 Objetivo do Trabalho

Diante desse cenário, o presente trabalho teve como objetivo elaborar um conjunto de ferramentas, códigos e materiais necessários para realizar o refinamento do modelo LLM-Compiler (CUMMINS et al., 2023), e possibilitar a aplicação futura de um treinamento adicional com programas e otimizações no domínio de microcontroladores, especificamente a plataforma alvo AVR do LLVM.

Nossa proposta envolveu a formação de uma base de dados específica, composta por códigos de representação intermediária (IR), extraídos durante o processo de compilação na linguagem ROBL. Estes códigos, previamente convertidos da linguagem C, juntamente com suas respectivas listas de passes ótimos de compilação, obtidas através de estratégias semi-exaustivas, são a base necessária para a elaboração de um *dataset* dedicado ao refinamento supervisionado do modelo de (CUMMINS et al., 2023). Nossa hipótese é que o refinamento permita que o modelo de IA seja capaz de compreender a estrutura e as restrições dos códigos voltados para MCUs AVR e, a partir disso, conseguir sugerir listas

personalizadas de passes de compilação.

Os objetivos específicos deste trabalho foram:

1. Construir uma base de dados contendo códigos em C e posteriormente convertê-los para ROBL;
2. Compilar essa base de dados utilizando Robcmp para códigos IR específicos para Atmega328p;
3. Encontrar os passes ótimos de instruções para cada código IR;
4. Elaborar sequências de *prompt/label* utilizando o IR não otimizado e os passes ótimos encontrados, suficientemente para formar um dataset expressivo;

1.3 Contribuição do Trabalho

Este trabalho contribui com a criação, elaboração e disponibilização de um pipeline completo para o treinamento e refinamento do LLM-Compiler no domínio de microcontroladores AVR. A contribuição inclui, de forma organizada e reprodutível:

- O desenvolvimento e documentação de um fluxo (pipeline) automatizado para geração, conversão e preparação de datasets compatíveis com o modelo (incluindo o conversor `c2rob` e os scripts de processamento);
- A construção e curadoria de datasets prontos, com códigos convertidos para ROBL, pares *prompt/label* adequados para treinamento supervisionado e metadados que permitam reproduzir experimentos;
- A disponibilização das ferramentas e artefatos necessários ao processo de refinamento (scripts de autotuning e mecanismos de validação e configuração para execução do fine-tuning na plataforma Hugging Face).

A contribuição dessa pesquisa foi definida como o desenvolvimento do processo completo e reprodutível que permite o refinamento: ou seja, todo o conjunto de métodos, datasets e ferramentas necessários para, posteriormente, executar o fine-tuning do LLM-Compiler no domínio AVR, bem como avaliar seus resultados. Assim, o mérito científico e prático deste trabalho reside na preparação e disponibilização de um pipeline robusto e pronto para ser usado no refinamento do modelo, reduzindo significativamente a barreira de entrada para futuras execuções experimentais e permitindo que trabalhos subsequentes concluam o fine-tuning de maneira reprodutível e eficiente.

2 Referencial Teórico

Este capítulo define os conceitos utilizados para a construção e estudo deste projeto. De início, conceitua-se compiladores e microcontroladores, o que são, como funcionam, onde atuam, suas limitações e capacidades. Posteriormente, o texto aborda Inteligência Artificial, detalhes de como funcionam, aprendizagem de máquina, em específico aprendizado supervisionado, LLMs e aplicações no mundo moderno. Por fim, este capítulo discute as possíveis estratégias de otimização de compiladores, com o uso de Inteligência Artificial.

2.1 Compilador

De acordo com Cooper e Torczon (2012), um compilador é um programa de computador que recebe outro programa em uma determinada linguagem de alto nível (linguagens de programação que se aproximam de linguagens humanas) e o traduz para uma linguagem de baixo nível (linguagem binária capaz de ser executada pelo processador do computador).

A importância de compiladores está atrelada ao desenvolvimento da tecnologia e à popularidade de sistemas de software em diversas aplicações. Com o advento da IoT (*Internet of Things*), conceito social que atrela a conexão de diversos dispositivos à rede e a computação na nuvem, a necessidade de compiladores otimizados se mostra maior do que nunca, visto que compiladores foram responsáveis pelo crescimento acelerado da tecnologia como conhecemos hoje (FISCHER; CYTRON; LEBLANC, 2010).

A estrutura básica de um compilador convencional pode ser definida em três etapas: *Front-end*, *Middle-end* e *Back-end* que seguem um fluxo de funcionamento decrescente desde o código de entrada mais alto nível até o código da máquina alvo, assim como ilustrado na Figura 1. O *Front-end* é responsável pela análise do código-fonte, convertendo-o em uma árvore sintática. Nessa etapa, ocorrem processos como a análise léxica, sintática e semântica, que verificam a estrutura e o significado do programa de acordo com as regras da linguagem de programação utilizada.

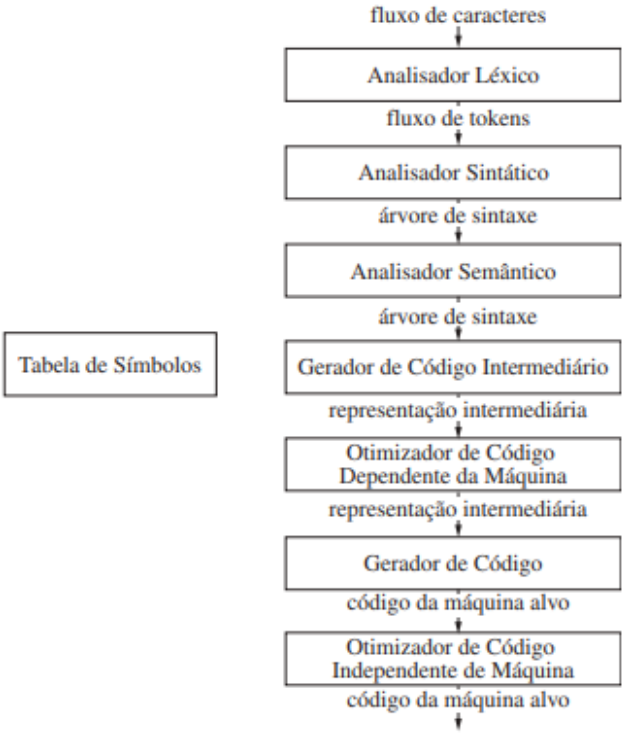

Figura 1 – Fases de um Compilador. Fonte: (AHO; SETHI; ULLMAN, 2007)

O *Middle-end* realiza otimizações independentes da arquitetura de hardware, transcrevendo o código original do programa de *input*(entrada) para uma versão intermediária otimizada de fácil discernimento, mantendo a semântica original do programa. O *Middle-end* pode ser representado como duas etapas que dependem da árvore sintática gerada pelo *front-end*, o gerador de código intermediário e o otimizador de código dependente da máquina alvo, ambas etapas geram ao final uma representação intermediária.

Por fim, o *Back-end* é encarregado de traduzir o código intermediário otimizado em código de máquina específico para a arquitetura alvo, com algumas alterações específicas para a melhor execução.

2.1.1 *Front-end*

O *front-end* é a etapa inicial do compilador. Ele é responsável por analisar o código-fonte e garantir que está bem estruturado para, posteriormente, poder ser traduzido em uma representação intermediária do mesmo programa. O *front-end* age de forma independente ao contexto da máquina aplicada, tendo uma estrutura de execução quase universal, aplicável para diferentes contextos de diferentes linguagens de alto nível (COOPER; TORCZON, 2012).

Sua abordagem consiste em analisar de forma léxica (o que é relativo ao vocabulário, à linguagem), depois analisar de forma sintática (o significado das expressões quanto à estrutura) e, por fim, analisar de forma semântica (o sentido da expressão no que tange

ao contexto em que está inserida). Dessa forma, o *front-end* consegue confirmar que o código-fonte é bem formado, escrito corretamente e possui sentido (COOPER; TORCZON, 2012).

2.1.2 Análise Léxica

O analisador léxico de um compilador é a primeira etapa percorrida do *front-end*. Essa parte, também conhecida como *Scanner*, tem como função transformar a sequência de caracteres do código-fonte em uma sequência de *tokens*, cada *token* representa uma unidade absoluta do código-fonte, como pontos, operadores, palavras-chave, números, símbolos, dentre outros. O *Scanner* é a primeira de três etapas que o compilador utiliza para entender o código-fonte, atuando diretamente sobre todo o código-fonte, fazendo com que essa etapa tenha um *input* maior do que as outras (COOPER; TORCZON, 2012).

O analisador léxico é construído com base em expressões regulares que definem os padrões léxicos da linguagem (COOPER; TORCZON, 2012) (FISCHER; CYTRON; LEBLANC, 2010). Essas expressões são transformadas em autômatos finitos que processam os caracteres um a um e identificam onde cada *token* começa e termina. Quando um *token* é reconhecido, ele é emitido com uma etiqueta indicando seu tipo e, em alguns casos, com um valor associado (como o nome de uma variável ou o valor numérico de uma constante).

Além de identificar *tokens*, o analisador léxico pode também eliminar elementos que não são importantes para a execução do código, como espaços em branco, formatações e comentários. Essa limpeza torna o processo mais eficiente e simplifica o trabalho do analisador sintático. Também é comum que o *scanner* registre a posição dos *tokens* (linha e coluna) para auxiliar na identificação do código segundo o formalismo das expressões dos autômatos finitos (COOPER; TORCZON, 2012).

2.1.3 Análise Sintática

O analisador sintático, também chamado de *parser*, é a segunda etapa do *front-end* de um compilador. Seu objetivo é verificar se a sequência de *tokens* produzida pelo analisador léxico está organizada de forma que respeite as regras gramaticais da linguagem de programação, geralmente descritas por uma gramática livre de contexto. Quando essa verificação tem sucesso, o parser constrói uma árvore sintática ou uma árvore de sintaxe abstrata (*Abstract Syntax Tree/AST*), que representa a estrutura hierárquica do programa de acordo com sua sintaxe. Conforme Fischer, Cytron e LeBlanc (2010), o parser é responsável por modelar a estrutura do programa de maneira que reflita sua lógica de construção, seu sucesso depende da definição cuidadosa da gramática da linguagem e da elaboração de seus algoritmos.

De acordo com Cooper e Torczon (2012), a análise sintática possui dois grandes

grupos de técnicas, são eles: *parsing top-down* e *parsing bottom-up* (*parsing* de cima para baixo e de baixo para cima, respectivamente), cada um possui técnicas diferentes (LL e LR), que devem ser escolhidas de acordo com a linguagem e com a complexidade. *Parsers* LL são mais simples e geralmente implementados manualmente, enquanto *Parsers* LR são mais poderosos e geralmente aplicados por outros softwares. Em qualquer técnica utilizada o resultado esperado é uma AST que representa de forma precisa o código fonte. A AST atua como uma espécie de conexão entre o *front-end* e o *back-end*, visto que servirá como a base para todas as demais análises e transformações que seguirão.

A Figura 2 ilustra uma AST para um trecho de código simples. Nela, cada nó representa uma construção sintática do programa, como uma declaração de variável, um identificador, um tipo, um operador, ou um inteiro literal. Por exemplo, a raiz da árvore pode ser uma função ou um bloco de código, que se desdobra em declarações e expressões. A estrutura hierárquica da AST reflete a precedência e a associação das operações no código original, por exemplo, ao representar a expressão $4 + 2 * 10 + 3 * (5 + 1)$, a AST mantém a ordem e a lógica original das funções do programa e remove detalhes irrelevantes da sintaxe (como parênteses ou pontos e vírgulas), focando apenas nos elementos essenciais para a semântica.

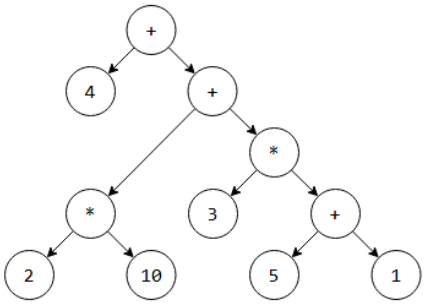

Figura 2 – Exemplo de uma *Abstract Syntax Tree* para $4 + 2 * 10 + 3 * (5 + 1)$. Fonte: <https://keleshev.com/abstract-syntax-tree-an-example-in-c/>

2.1.4 Análise Semântica

O analisador semântico é a terceira etapa do *front-end* de um compilador, responsável por garantir que o programa, além de estar correto do ponto de vista sintático, seja também semanticamente coerente. Ou seja, ele verifica se o programa faz sentido no contexto da linguagem, respeitando regras como declaração e uso de variáveis, estrutura de dados e outros aspectos que não podem ser descritos apenas por uma gramática livre de contexto. Essa análise é feita a partir da AST gerada pelo *parser*, geralmente com o auxílio de uma tabela de símbolos (COOPER; TORCZON, 2012).

Segundo Fischer, Cytron e LeBlanc (2010), a análise semântica percorre a AST executando diversas verificações contextuais que não podem ser descritas apenas por uma

gramática livre de contexto. Essas verificações são realizadas em etapas bem definidas, que compõem o fluxo típico dessa fase no compilador:

1. **Construção e manutenção da tabela de símbolos**: à medida que a AST é percorrida, o compilador coleta informações sobre declarações de variáveis, funções, tipos e escopos, armazenando-as na tabela de símbolos para posterior consulta e validação (COOPER; TORCZON, 2012).

2. **Verificação de tipos**: o analisador semântico assegura que as expressões e operações do programa estejam de acordo com as regras de tipagem da linguagem. Por exemplo, operadores aritméticos devem atuar sobre operandos numéricos compatíveis, e chamadas de função devem passar argumentos do tipo e quantidade corretos (FISCHER; CYTRON; LEBLANC, 2010).

3. **Verificação de conversão de tipos (*casting*)**: o compilador analisa se há necessidade de conversão entre tipos, automática ou explícita, e verifica se essas conversões são válidas no contexto da linguagem. Erros ou alertas são gerados quando conversões perigosas não são tratadas corretamente (AHO; SETHI; ULLMAN, 2007).

4. **Verificação de escopo**: a análise semântica valida se as variáveis e funções estão sendo acessadas dentro de seus escopos apropriados. Isso evita, por exemplo, o uso de variáveis declaradas fora do bloco de código atual.

5. **Verificação de fluxo de controle**: comandos como `break`, `continue` e `return` são analisados para garantir que aparecem em contextos válidos, como dentro de laços ou funções.

6. **Verificação de unicidade**: o compilador verifica se nomes de variáveis, membros de estruturas, funções ou rótulos não estão sendo definidos mais de uma vez dentro do mesmo escopo, o que seria ilegal.

Cada uma dessas etapas contribui para assegurar que o programa respeite as regras contextuais da linguagem de programação. Ao final dessa fase, a AST encontra-se enriquecida com informações de tipo e escopo, e pronta para ser traduzida para uma representação intermediária, iniciando a próxima etapa do processo de compilação (AHO; SETHI; ULLMAN, 2007).

2.1.5 Geração de código intermediário

Uma das últimas coisas que o *front-end* de um compilador faz, é a geração de código intermediário (AHO; SETHI; ULLMAN, 2007). Essa representação intermediária (IR, do inglês *Intermediate Representations*), geralmente utilizada para otimizações e para facilitar

a geração do código de máquina (COSTA et al., 2023), serve como entrada para o gerador de código, que, juntamente com as informações da tabela de símbolos, produz o código objeto equivalente. As IRs são a forma de como um compilador consegue representar os estados e as informações do código que ele compila, compiladores podem utilizar uma ou mais representações intermediárias, variando de acordo com o processo executado para a tradução até a linguagem alvo (COOPER; TORCZON, 2012).

Existem diversos tipos de Representações Intermediárias (IRs) utilizadas em compiladores, cada uma com suas características e finalidades específicas (COOPER; TORCZON, 2012). Alguns compiladores geram uma representação intermediária de baixo nível chamada, código de três endereços, nessa IR, cada instrução contém no máximo uma operação, e devido a sua simplicidade é mais simples aplicar otimização de código nesta representação, essa representação intermediária deve ter duas propriedades importantes: ser facilmente produzida e ser facilmente traduzida para a máquina alvo (AHO; SETHI; ULLMAN, 2007).

Como o LLM-Compiler é voltado para arquiteturas de computadores que utilizam a infraestrutura LLVM, o foco deste trabalho será no LLVM-IR, a representação intermediária de compiladores a base de LLVM, projetada para ser uma "IR universal", capaz de representar diversas linguagens de alto nível (The LLVM Project, 2025). A LLVM-IR pode ser dividida entre: uma representação em memória para o compilador, uma representação em *bitcode* para carregamento rápido e uma representação em linguagem *assembly* legível por humanos. Essa abrangência da LLVM-IR permite processos de otimização eficiente para transformações do compilador, além de detectar possíveis erros ou problemas na tradução.

A Figura 3 é uma representação de código intermediário LLVM, suas instruções como 'zext', 'trunc', 'br', 'call' e 'ret' exemplificam operações típicas da linguagem intermediária, como extensão de zero, truncamento de inteiros, desvios condicionais e incondicionais, chamadas de função e retornos, sua linguagem compreensível para humanos permite fácil compreensão de código e facilita a análise, otimização e posterior uso. Essa clareza é especialmente útil durante o desenvolvimento e treinamento de modelos baseados em aprendizado de máquina, como o LLM-Compiler de Cummins et al. (2023), que dependem da extração de padrões sintáticos e semânticos do LLVM-IR para sugerir ou otimizar sequências de passes de compilação.

2.1.6 *Middle-end*

O *Middle-end* de um compilador atua como uma ponte entre a análise (*front-end*) e a geração de código final (*back-end*). Seu principal objetivo é transformar a árvore de sintaxe abstrata (AST), produzida e validada pelas fases anteriores, juntamente com a tabela de símbolos, em uma representação intermediária (*Intermediate Representation/IR*)

```
target triple="x86_64"

%struct.list = type { i64, %struct.list* }
%struct.mystruct = type { double, i32, [10 x i8] }

@globallist = common global %struct.list* null, align 4

define void @tailrecursive(i64 %num)
{
LU2:
  br label %LU3
LU3:
  %u0 = icmp sle i64 %num, 0
  %u1 = zext i1 %u0 to i64
  %u2 = trunc i64 %u1 to i1
  br i1 %u2, label %LU5, label %LU4
LU5:
  ret void
LU4:
  %u5 = sub i64 %num, 1
  call void @tailrecursive(i64 %u5)
  ret void
}
```

Figura 3 – Exemplo de um código em LLVM-IR. Fonte: <https://github.com/colejcummins/llvm-syntax-highlighting>

de forma que seja mais fácil para otimizar e depois gerar o código em linguagem baixo nível (COOPER; TORCZON, 2012).

Uma boa representação intermediária abstrai os detalhes da linguagem fonte e da máquina alvo, permitindo que o compilador aplique uma série de análises e otimizações, como propagação de constantes, eliminação de código morto, análise de alcance de variáveis e redução de laços (FISCHER; CYTRON; LEBLANC, 2010). Essas transformações visam melhorar o desempenho, reduzir o consumo de memória e aumentar a eficiência do código gerado, sem alterar o comportamento do programa original.

Além disso, o uso de uma IR padronizada, como o SSA (*Static Single Assignment*), proporciona uma base sólida para a construção de compiladores capazes de gerar código para diferentes arquiteturas a partir do mesmo núcleo de análise (FISCHER; CYTRON; LEBLANC, 2010). Um bom exemplo é a LLVM (*Low Level Virtual Machine*) que adota o formato SSA como estrutura central em sua representação intermediária, o LLVM-IR. Essa escolha permite que múltiplas ferramentas de otimização e geração de código operem de forma coesa e eficiente sobre uma representação comum. Essa separação entre as fases do compilador, favorece a reutilização de componentes, a portabilidade e a manutenção do sistema de compilação (COOPER; TORCZON, 2012).

É também no *Middle-end* que ocorre o processamento dos passes de otimização de código independentes de plataforma alvo, os quais serão alvo de escolha e ordenação pelo modelo de IA a ser refinado neste trabalho.

2.1.7 *Back-end*

O *back-end* de um compilador, segundo Fischer, Cytron e LeBlanc (2010) e Cooper e Torczon (2012), é responsável por transformar a representação intermediária (IR) produzida pelo *front-end* em código de máquina ou código objeto específico para a arquitetura alvo. Essa fase ocorre após todas as análises e validações semânticas terem sido realizadas, e o programa já estar representado de forma estruturada e otimizada.

De acordo com Cooper e Torczon (2012), o *back-end* executa três funções principais: seleção de instruções, alocação de registradores e agendamento de instruções. A seleção de instruções envolve mapear operações da IR para instruções da linguagem de máquina. Já a alocação de registradores trata da distribuição eficiente das variáveis temporárias nos registradores físicos disponíveis, dado o número limitado desses recursos. O agendamento de instruções busca reorganizar o código para melhorar a performance, por exemplo, evitando dependências e aumentando o paralelismo entre instruções.

Fischer, Cytron e LeBlanc (2010) destacam que o objetivo do *back-end* é gerar código eficiente e correto para a máquina alvo, respeitando todas as restrições impostas pela arquitetura. Ele também ressalta a importância da modularidade: ao usar uma representação intermediária bem projetada, torna-se possível desenvolver diferentes *back-end* para várias arquiteturas, reutilizando o *front-end* e o *middle-end*. Isso é especialmente útil para compiladores que precisam ser portáveis ou suportar múltiplas plataformas.

2.2 Microcontroladores

Microcontroladores são componentes fundamentais no desenvolvimento de sistemas embarcados. De forma geral, um microcontrolador pode ser definido como um processador que possui recursos integrados, tais como memória RAM, espaço para código e interfaces periféricas, como linhas de entrada e saída (WHITE, 2024). Essa integração de funcionalidades permite que os microcontroladores operem de forma independente em aplicações específicas, sem a necessidade de sistemas operacionais completos, diferentemente dos computadores de uso geral (HUSSAIN et al., 2016).

O uso de microcontroladores está amplamente difundido em dispositivos do cotidiano, como eletrodomésticos, brinquedos eletrônicos, sistemas automotivos, equipamentos médicos, sensores industriais, entre outros. Por serem aplicados em contextos de objetivo restrito, esses dispositivos normalmente enfrentam limitações significativas em termos de recursos computacionais, consumo de energia, e capacidade de armazenamento (WHITE, 2024). Um bom exemplo disso é o modelo de MCU ATmega328p mostrado na Figura 4, pertencente a família de microcontroladores AVR, alvo deste trabalho. Este chip implementa uma arquitetura Advanced RISC de 8 bits e possui somente 256KB de memória *flash* programável (Atmel Corporation, 2015).

Figura 4 – Foto de exemplificação do microcontrolador estudado, Atmega328p Fonte: <https://www.conectabit.com.br/MLB-4181008568-chip-atmega328p-atmega328p-u-atmega-328-pu-dip-32k-20mhz-_JM>

Segundo White (2024), a programação para microcontroladores é feita diretamente sobre o *bare metal*, ou seja, sem a presença de um sistema operacional intermediário. Isso significa que, ao escrever um comando como "acender um LED", o software se comunica diretamente com o hardware, conferindo maior controle e eficiência, características essenciais em sistemas com restrições severas de tempo real e consumo.

Além disso, microcontroladores geralmente estão acoplados a diversos sensores, atuadores e demais periféricos, sendo necessário lidar com a integração entre hardware e software de forma coordenada e robusta. Para enfrentar tais desafios, boas práticas de arquitetura de software, como modularidade, encapsulamento e uso de padrões de projeto, são essenciais para garantir flexibilidade, manutenção e reusabilidade do sistema ao longo de seu ciclo de vida (WHITE, 2024).

2.2.1 *Domain Specific Languages*(DSL)

As *Domain-Specific Languages* (DSLs) são linguagens de programação ou notação projetadas para atender a um domínio específico de aplicação, oferecendo maior expressividade e facilidade de uso nesse contexto (MERNIK; HEERING; SLOANE, 2005). Ao contrário das linguagens de programação de uso geral, que tentam atender a uma ampla variedade de problemas, as DSLs focam em um nicho restrito, como bancos de dados, planilhas eletrônicas, modelagem de *hardware* ou geração de relatórios.

Justamente pelo fato de serem desenvolvidas com um domínio específico em mente, essas linguagens tornam o trabalho mais simples e direto. Elas permitem, por exemplo, que profissionais da área possam interagir com sistemas complexos de forma mais intuitiva, sem necessitar de um alto grau de qualificação técnica (MERNIK; HEERING; SLOANE, 2005). Essa especialização pode reduzir os custos de desenvolvimento e manutenção, ao mesmo tempo que aumenta a reutilização de *software* e a precisão dos sistemas (MERNIK; HEERING; SLOANE, 2005).

Além disso, DSLs podem ser implementadas de diferentes formas, como interpretadores, compiladores, pré-processadores ou linguagens embutidas. Essa flexibilidade de implementação e a capacidade de abstrair detalhes de baixo nível e expressar intenções de

forma mais clara e concisa é particularmente vantajosa no contexto de sistemas embarcados, que operam sob severas restrições de memória, tempo de execução, consumo de energia e capacidade de processamento. De acordo com White (2024), práticas como modularidade, encapsulamento e reutilização de componentes são fundamentais para enfrentar os desafios comuns no desenvolvimento de *software* embarcado. DSLs bem projetadas podem fortalecer esses princípios ao oferecer construções específicas que se alinham diretamente com o hardware e com as tarefas que o software precisa desempenhar.

2.2.2 Robotics Language

A *Robotics Language* (ROBL) é uma linguagem de programação e compilador desenvolvidos com foco em aplicações de microcontroladores voltadas para robótica e Internet das Coisas (IoT). Seu principal diferencial está na abstração das particularidades de hardware diretamente na linguagem e em sua biblioteca padrão, permitindo que o mesmo código-fonte seja compilado para diferentes plataformas sem necessidade de ajustes manuais ou uso de diretivas condicionais (OLIVEIRA, 2024).

Além de promover portabilidade entre arquiteturas, o ROBL realiza uma análise semântica aprofundada durante a compilação, prevenindo diversos tipos de erros que normalmente só seriam identificados em tempo de execução. Essa característica contribui para um desenvolvimento mais seguro e confiável, especialmente em sistemas embarcados, onde falhas podem ser críticas.

A implementação do compilador foi construída utilizando ferramentas clássicas no desenvolvimento de compiladores, como o *Flex* (versão 2.6.4) para análise léxica e o *Bison* (versão 3.8.2) para análise sintática e geração de código intermediário. O *backend* do compilador se apoia no *LLVM*, um *framework* moderno e modular que possibilita tanto otimizações quanto a geração de código para diferentes arquiteturas de microcontroladores.

O ecossistema do Robcmp também inclui suporte a depuração via simulador e integração com o Visual Studio Code (editor de código), por meio da extensão *RobCmpSyntax*, que fornece realce de sintaxe e facilita o desenvolvimento. Esses recursos tornam o Robcmp uma ferramenta especialmente atrativa em ambientes educacionais, oferecendo uma alternativa mais segura e acessível ao tradicional uso da linguagem C em projetos embarcados.

2.3 Inteligência Artificial

De acordo com Russell, Russell e Norvig (2020) a Inteligência Artificial (IA) pode ser compreendida como o estudo de agentes inteligentes, entidades que percebem seu ambiente e tomam ações que maximizam suas chances de sucesso em atingir objetivos. Para que algo seja considerado uma IA, ele deve ser capaz de realizar funções geralmente feitas por seres

humanos, como perceber o ambiente, raciocinar e tomar decisões, aprender com dados e agir no mundo físico (MORANDÍN-AHUERMA, 2022; BODEN, 2017). Seja através da robótica, do aprendizado de máquina, de sistemas probabilísticos ou de mapeamento 3D em tempo real, softwares de Inteligência Artificial possuem a capacidade de agir de forma racional ou até imitar a capacidade humana até certo ponto (MORANDÍN-AHUERMA, 2022).

Recentemente a IA tem-se demonstrado uma ferramenta poderosa para trabalho, logística e lazer dentro de diversos setores da sociedade atual (LUDERMIR, 2021). Tecnologias como reconhecimento de voz, tradução automática, veículos autônomos, sistemas de recomendação e diagnósticos médicos automatizados são somente alguns exemplos de como algoritmos e softwares de Inteligência Artificial são necessários e fazem parte do mundo moderno (RUSSELL; RUSSELL; NORVIG, 2020; LUDERMIR, 2021). A característica singular de adaptação de forma racional a ambientes complexos e incertos, antes presente somente em humanos, evidencia a importância desses sistemas inteligentes no desenvolvimento de novas tecnologias e na superação de desafios anteriormente insuperáveis (MORANDÍN-AHUERMA, 2022).

No nível mais geral, uma IA funciona como um agente racional, que percebe seu ambiente por meio de sensores e age sobre ele por meio de atuadores. Esse agente processa sequências de percepções para decidir qual ação tomar, com base em alguma função desejada. O agente pode ser simples, reagindo diretamente aos estímulos, ou complexo, utilizando modelos internos, planejamento, raciocínio e aprendizado para adaptar seu comportamento a longo prazo (RUSSELL; RUSSELL; NORVIG, 2020; MUHAMMAD; YAN, 2015).

2.3.1 *Machine Learning*

Russell, Russell e Norvig (2020) explicam que *machine learning* é uma subárea da Inteligência Artificial que estuda algoritmos e modelos que permitem a sistemas computacionais aprender com base em experiências, grupos de dados e diretrizes, para melhorar sua habilidade e execução de alguma tarefa. Em outras palavras é campo da IA que estuda como agentes podem melhorar automaticamente seu desempenho por meio da experiência (MUHAMMAD; YAN, 2015; MORANDÍN-AHUERMA, 2022).

De uma forma didática, *machine learning* pode ser explicado pela seguinte lógica: diferentemente de outros sistemas e softwares com execução linear, onde se tem o *input*(entrada), o método de processamento e se deseja saber a saída, em algorítmos de *machine learning* se posssui o *input* e o *output*(saída) e se deseja saber ou fazer com que a máquina entenda, o meio termo, ou seja, o caminho até a saída. Na prática, em vez de programar explicitamente todas as regras para uma tarefa, fornece-se ao sistema uma grande quantidade de dados para que ele interprete padrões e regras por conta própria,

podendo replicá-los futuramente (RUSSELL; RUSSELL; NORVIG, 2020; LUDERMIR, 2021).

Atualmente, qualquer treinamento de IA envolve três elementos principais: os exemplos (dados de entrada e saída esperada), o modelo de representação e a função de avaliação e otimização. Os exemplos são o que alimenta a aprendizagem, são os dados iniciais que o modelo de representação recebe durante sua formação como IA, podem ser diversos tipos de arquivos ou outras formas de dados. O modelo de representação é a forma matemática ou computacional usada para expressar o conhecimento aprendido, pode ser uma árvore de decisão binária ou um sistema de nós que manipula informações (redes neurais). Por final a função de otimização avalia os resultados obtidos pelo modelo durante o treinamento e corrige seus parâmetros internos para melhorar os resultados com base em critérios de erro ou recompensa.

Dessa forma, Russell, Russell e Norvig (2020) define que o tipo de *machine learning* aplicado a um modelo depende da natureza da tarefa e dos dados disponíveis para treiná-lo. Podendo serem divididos entre:

1. Aprendizado supervisionado, dados rotulados que ajudam o modelo a classificar semelhanças entre os dados em grupos dos rótulos facilitando o reconhecimento de padrões. Ex: Fotos de pássaro com o rótulo papagaio e fotos de árvores com o rótulo pinheiro, facilitariam um modelo de reconhecimento de imagens.

2. Aprendizado não supervisionado, dados não rotulados, permitindo ao modelo descobrir padrões não específicos nos exemplos, facilitando conexões não aparentes. Ex: Agrupamento de arquivos de diferentes tipos com base no conteúdo.

3. Aprendizado por reforço, um agente recebe recompensas conforme os resultados que suas ações geram no meio inserido. Ex: Ações de um boneco ao chegar mais longe em uma fase de um jogo eletrônico.

2.3.2 Aprendizado Supervisionado

Com base no que Russell, Russell e Norvig (2020) o aprendizado supervisionado é uma das formas mais fundamentais e amplamente utilizadas de *machine learning*. Nesse tipo de aprendizado, o agente recebe um conjunto de exemplos rotulados, ou seja, entradas acompanhadas de suas respectivas saídas corretas. O objetivo do sistema é aprender uma função que generalize esses exemplos, ou seja, que seja capaz de prever corretamente a saída para novas entradas nunca vistas antes. O conjunto de exemplos é composto por pares de dados (*data*,*label*), esses pares serão analisados por uma árvore de decisão que compõe o sistema de aprendizado, permitindo que o modelo identifique padrões e classifique de forma precisa durante a execução (MUHAMMAD; YAN, 2015).

Durante o treino cada exemplo de informa a resposta certa, e o algoritmo tenta ajustar seu modelo interno para minimizar os erros entre suas previsões e os valores reais. Com o tempo e com mais exemplos, o modelo vai melhorando sua capacidade de prever resultados corretos mesmo quando confrontado com dados novos (RUSSELL; RUSSELL; NORVIG, 2020). A Figura 5 demonstra esse processo. O modelo de IA é alimentado por fotos de corujas, raposas e esquilos e também pelos rótulos respectivos de cada foto, compondo assim os pares (*data*,*label*), então o modelo é capaz de aprender regras e funções relativas a esse conjunto de dados que permite que ele classifique corretamente as imagens após terminado o treinamento.

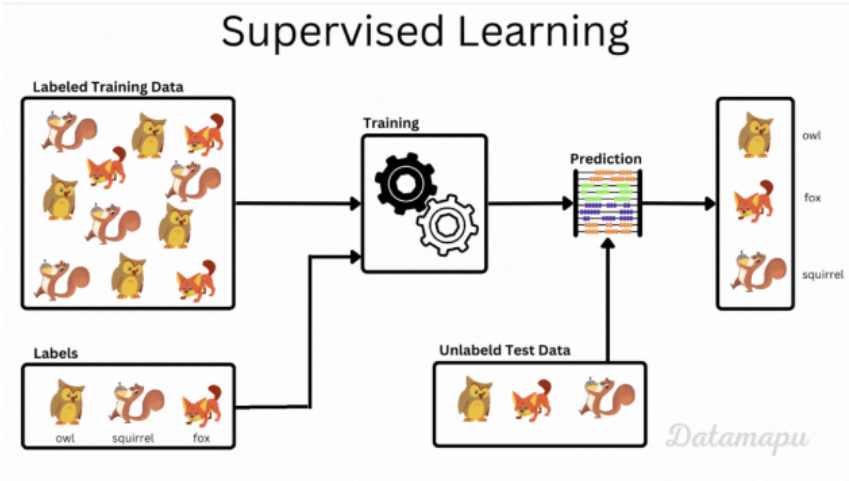

Figura 5 – Representação didática do funcionamento de aprendizagem supervisionado. Fonte: <https://datamapu.com/posts/ml_concepts/supervised_unsupervised>

Existem diversos exemplos de aprendizado supervisionado, os mais famosos são tarefas como classificação e regressão. Na classificação, o objetivo é prever uma categoria discreta, como determinar se uma foto é um papagaio ou uma árvore. Já na regressão, a saída é um valor contínuo, como prever o preço de uma casa com base em suas características (tamanho, localização, etc.).

Modelos comumente usados para aprendizado supervisionado incluem árvores de decisão, redes neurais, modelos lineares, entre outros. Cada um deles tem suas vantagens e é mais adequado para certos tipos de dados ou problemas. Independentemente do modelo escolhido, o desempenho é geralmente avaliado usando conjuntos de dados separados de teste, medindo métricas como acurácia, precisão, recall ou erro quadrático médio, dependendo da tarefa.

2.3.3 *Large Language Models (LLM)*

Um LLM, ou um grande modelo de linguagem, é um tipo de modelo estatístico treinado para prever a próxima palavra em uma sequência de texto, dado o contexto

anterior. Essa tarefa, chamada de modelagem de linguagem, é um problema de aprendizado supervisionado (ou auto-supervisionado) onde, para cada entrada (um fragmento de texto), o modelo deve prever a saída correta (a próxima palavra ou *token*). Durante o treinamento, ele processa bilhões de palavras, ajustando seus bilhões de parâmetros para capturar padrões linguísticos, sintaxe, semântica e até aspectos pragmáticos do uso da linguagem (CHANG et al., 2024).

O funcionamento básico de um LLM envolve três fases: o pré-treinamento, onde o modelo aprende representações gerais da linguagem com base em enormes volumes de texto (livros, sites, artigos); a refinamento (*fine-tuning*), em que ele é ajustado para tarefas específicas com dados mais controlados; e, em alguns casos, a aplicação de técnicas como aprendizado por reforço com *feedback* humano (RLHF), que serve para alinhar as respostas do modelo a valores humanos, como segurança, utilidade e cortesia (ZHAO et al., 2024).

LLMs são construídos sobre arquiteturas de redes neurais profundas chamadas *transformers*, que permitem o processamento eficiente de sequências longas de texto, capturando relações contextuais entre palavras, mesmo que estejam distantes no texto. Essa tecnologia presente na arquitetura de modelos de linguagem é fundamental para o funcionamento de paralelismo durante o treinamento do modelo, e também para permitir maior escalabilidade para maiores conjuntos de dados (VASWANI et al., 2017; CHANG et al., 2024).

O treinamento de LLMs como ChatGPT[1], Gemini[2] e Llama 2[3] se dá graças a um pré-treinamento e refinamento exaustivos e complexos, envolvendo um processo iterativo e em larga escala que se estende por várias etapas. Inicialmente, esses modelos passam por um pré-treinamento extensivo com um volume enorme de dados textuais, devido a isso o treinamento dessas IAs é extremamente custoso e demorado.

2.4 Otimização em Compiladores Modernos

A etapa de otimização em compiladores visa transformar o código intermediário (IR – *Intermediate Representation*) em uma versão mais eficiente, sem alterar sua semântica. Essa eficiência pode se referir à melhoria no desempenho de execução, economia de memória, redução de tamanho do binário, ou mesmo economia de energia, dependendo dos objetivos da aplicação alvo (COOPER; TORCZON, 2012; FISCHER; CYTRON; LEBLANC, 2010).

Tradicionalmente, compiladores como o LLVM aplicam conjuntos padronizados de passes de otimização, agrupados em níveis como `-O1`, `-O2` e `-O3`, com sequências específicas voltadas à performance, e `-Os` ou `-Oz`, voltadas à compactação do código. Esses conjuntos

[1] Site dos modelos do ChatGPT: <https://platform.openai.com/docs/models>
[2] Site dos modelos do Gemini: <https://ai.google.dev/gemini-api/docs/models?hl=pt-br>
[3] Site da Meta para Llama2: <https://www.llama.com/llama2/>

são compostos por dezenas ou até centenas de transformações e análises estáticas, incluindo eliminação de código morto, propagação de constantes, *inlining* e ordenação de instruções (LATTNER; ADVE, 2004).

Apesar da eficácia dessas abordagens, elas não são adaptáveis ao perfil específico de cada programa. Por isso, nos últimos anos, pesquisadores têm investigado o uso de aprendizado de máquina para guiar decisões de otimização de forma mais personalizada. A ideia central é que modelos possam aprender, a partir de exemplos anteriores, quais sequências de otimização produzem melhores resultados para diferentes classes de programas (WANG; O'BOYLE, 2018).

Um dos desafios centrais dessa abordagem é representar programas de forma mensurável. A extração de *features* — características numéricas derivadas da estrutura e comportamento do código — é essencial para alimentar modelos preditivos. Em trabalhos recentes, propõe-se representar programas como grafos, como forma de capturar dependências de controle e dados, permitindo que modelos de aprendizado relacional compreendam melhor o contexto das instruções (CUMMINS et al., 2023).

Recentemente, modelos de linguagem de larga escala (LLMs) têm sido aplicados ao problema de otimização. Esses modelos são capazes de inferir boas sequências de passes de forma contextual, generalizando a partir de grandes volumes de dados de código e otimizações anteriores. O trabalho de Cummins et al. (2023) demonstra que LLMs treinados com representações intermediárias, como o LLVM-IR, podem alcançar desempenho competitivo com métodos heurísticos clássicos, muitas vezes sem a necessidade de compilações repetidas ou ajustes manuais extensivos.

Portanto, a otimização de compiladores pode, no futuro, passar por um momento de transição de heurísticas generalistas para abordagens baseadas em dados, aprendizado e raciocínio automatizado. Essa mudança combina os princípios tradicionais descritos por Cooper e Torczon (2012) e Fischer, Cytron e LeBlanc (2010) com os avanços em IA e aprendizado de máquina discutidos por Russell, Russell e Norvig (2020), abrindo caminho para compiladores mais inteligentes e adaptáveis.

3 Trabalhos Relacionados

3.1 Introdução

Este capítulo apresenta uma revisão acerca dos principais trabalhos relacionados ao tema desta pesquisa, com foco nas áreas de otimização de tamanho de código, busca de sequências de passes de compilação e, em especial, técnicas envolvendo modelos de linguagem de larga escala. O objetivo é contextualizar o refinamento do modelo LLM-Compiler para o domínio de microcontroladores AVR dentro do escopo da Robotics Language (ROBL), destacando contribuições, limitações e relações diretas com este estudo.

Inicialmente, são apresentados os critérios adotados para seleção dos trabalhos. Em seguida, são descritas as obras consideradas mais relevantes, discutindo suas metodologias, resultados e pertinência para o desenvolvimento desta pesquisa.

3.2 Metodologia de Análise

A análise dos trabalhos relacionados seguiu uma abordagem de Revisão Narrativa de Literatura (RNL), com busca realizada para quaisquer trabalhos que de forma direta ou indiretamente abordaram otimização de tamanho de código, associados ao ambiente LLVM. A seleção dos materiais também considerou referências citadas no próprio trabalho original do LLM-Compiler, em especial pesquisas que envolvem:

- **Modelos de IA aplicados à otimização de compiladores**;
- **Métodos de busca e seleção de passes de otimização** para redução de tamanho de código;
- **Ferramentas tradicionais de autotuning** como YaCoS;
- **Redução sistemática de sequências de passes**;
- **Construção de datasets** baseados em representações intermediárias;

3.3 Trabalhos Analisados

Com base nos critérios definidos, três trabalhos foram considerados fundamentais para o desenvolvimento desta pesquisa, por suas contribuições conceituais e práticas no problema de otimização de código e seleção de passes de compilação. As subseções a seguir apresentam um resumo estruturado de cada obra, destacando suas contribuições e sua

relação direta com a estrutura desta pesquisa e seu impacto em um futuro refinamento do modelo LLM-Compiler para o domínio AVR.

3.3.1 LLM-Compiler

O artigo de Cummins et al. (2023) introduz o LLM-Compiler, a primeira aplicação extensiva de Large Language Models ao problema de ordenação de passes do LLVM com objetivo de redução de tamanho de código. Os autores treinam um modelo, utilizando a biblioteca transformers, com 7 bilhões de parâmetros e arquitetura LLaMA-2 a partir do zero, usando um conjunto de 1 milhão de funções LLVM-IR, e estabeleceram que além de prever a lista de passes, o modelo também aprende a predizer a contagem de instruções antes e depois de gerar o IR otimizado como tarefa auxiliar.

De forma prática, os autores constroem um padrão de sequências via autotuning (busca extensiva e algoritmo de minimização das sequências), e usam esse padrão como label supervisionado para treinar o LLM. O treinamento envolveu aproximadamente 15.7 bilhões de tokens e 620 dias de GPU, como referência, o autotuner alcançava ganhos maiores (5%) mas à custa de bilhões de compilações adicionais, enquanto o LLM atingiu um ganho prático de 3% sobre a flag `-Oz` sem invocar o compilador em inferência. Complementarmente, o modelo produz código compilável em 90% dos casos e acerta o IR final caractere por caractere em cerca de 70%, evidenciando capacidade real de raciocínio sobre transformações de compilador.

Para este trabalho, Cummins et al. (2023) fornecem o fundamento conceitual e base prática (o próprio LLM-Compiler). Os autores provaram que LLMs podem aprender lógicas de otimização e substituir buscas ineficientes por predições eficientes. Sua pesquisa, porém, foi elaborada para arquiteturas de propósito geral e com janelas de contexto e datasets dimensionados para esse tipo de CPU. Este presente trabalho, aproveita essa abordagem mas a adaptada ao domínio de MCU AVR (em particular ATmega328P), enfrentando desafios diferentes, como severas restrições de memória, conjunto de instruções reduzido, necessidade de converter e mapear passes entre versões do LLVM e limites de tempo, por isso, a maior contribuição da pesquisa de Cummins et al. (2023) para este trabalho é a base teórica e demonstração prática que inspirou a necessidade da atual pesquisa.

3.3.2 Good Optimization Sequences Covering Program Space

O trabalho de Purini e Jain (2013) propõe um método para identificar sequências de otimização que cobrem eficientemente o espaço de programas, reduzindo a necessidade de testes exaustivos em todos os códigos de entrada. A abordagem utiliza remoção iterativa e análise de impacto dos passes, permitindo identificar sequências curtas, porém eficazes, na redução de tamanho e melhoria de desempenho.

Esta pesquisa é fundamental para o presente trabalho, pois seu método de redução de sequências é utilizado diretamente no processo de filtragem das listas geradas pelo autotuning descrito por Faustino et al. (2021) e utilizado na lista *Optimization Cache*. Assim, sua contribuição teórica fornece a base para garantir que somente os passes essenciais sejam mantidos nas listas que farão parte do dataset final.

3.3.3 Optimization Sequences for Code-Size Reduction

O trabalho de Faustino et al. (2021) apresenta um método sistemático para encontrar sequências de otimização voltadas especificamente à redução de tamanho de código no LLVM. Os autores realizaram um estudo em grande escala, compilando 15 000 programas com 10 044 sequências distintas geradas e avaliadas, usando um algoritmo genético (via YaCoS) para produzir um conjunto inicial de candidatas, seguido da aplicação do algoritmo de redução de Purini e Jain (2013) para eliminar passes redundantes. Desse processo emergiram cinco sequências curtas (12–15 transformações) que, em vários benchmarks, chegaram a produzir binários menores que as flags padrão do LLVM e reduziram o tempo de compilação em relação a `-Os`/`-Oz`. Essas realizações estão detalhadas no estudo e servem como demonstração prática de que sequências bem escolhidas e compactas podem chegar perto, e em alguns casos superar, as configurações padrão, com ganho também em tempo de compilação.

Do ponto de vista metodológico, o trabalho organiza critérios úteis para selecionar e avaliar sequências: a construção da matriz programa sequência (*optimization matrix*), medidas como "*best frequency*", "*good in bounds*", "*total size*", "*geometric relative size*"e "*best maximum relative size*". Essas ferramentas permitem quantificar não apenas qual sequência é "melhor" para um programa isolado, mas também analisar robustez e diversidade de comportamento sobre um corpus extenso. Além disso, os resultados mostram que as novas sequências tendem a ser muito mais curtas que `-Os`/`-Oz` (média aproximadamente 40 a 79 passes nas propostas versus 260 nas flags), o que explica ganhos substanciais em tempo de compilação e qualidade ou compacidade em muitos casos.

Para o presente TCC, a influência do trabalho de Faustino et al. (2021) é direta e prática: adoptamos a *Optimization Cache* como fonte principal de sequências candidatas e como ponto de partida para a busca da melhor sequência por programa. A abordagem deles justificou tanto a escolha de testar um conjunto finito e criterioso de sequências (em vez de tentar uma busca exponencial) quanto a aplicação do processo de redução e filtragem para obter sequências compactas e reprodutíveis, mostrando que o método deles abrange um caminho aplicável, embora exija mudanças adicionais para domínios com restrições severas de recursos.

4 Metodologia do desenvolvimento

4.1 Introdução

Neste capítulo será abordado o método aplicado no escopo desta pesquisa. Será detalhado a abordagem empregada durante as diversas etapas do processo de elaboração deste trabalho, especificando pontos chaves de cada etapa para garantir a replicabilidade desta pesquisa. Neste capítulo, também será exemplificado e descrito os códigos e ferramentas, utilizadas durante todo o processo, que podem ser encontrados neste repositório do Github.

4.2 Elaboração do conjunto de códigos C

A primeira etapa do pipeline consiste na geração do conjunto inicial de programas em linguagem C, a partir de duas fontes distintas: códigos sintéticos produzidos automaticamente e códigos reais obtidos de repositórios públicos. O objetivo desta etapa é formar uma base ampla, variada e representativa o suficiente para sustentar o processo posterior de conversão, compilação e autotuning.

4.2.0.1 Códigos sintéticos gerados com Csmith

Para a geração de códigos sintéticos, foi utilizado o software Csmith (YANG et al., 2011), disponibilizado publicamente em <https://github.com/csmith-project/csmith>. Após a instalação da ferramenta conforme instruções do repositório oficial, iniciou-se um processo exploratório para determinar parâmetros adequados de geração. Foram realizados diversos testes com diferentes combinações de opções do Csmith, a fim de identificar uma configuração capaz de produzir códigos: Não excessivamente extensos, evitando que os programas ultrapassassem a janela de contexto do modelo de linguagem a ser futuramente treinado e não demasiadamente simples, de modo a preservar diversidade estrutural e evitar um conjunto de dados pobre em variabilidade, que potencialmente conduzisse a *overfitting*.

Uma vez definidos os parâmetros ideais, foi desenvolvido um script em *shell*, denominado `csmithrun.sh`, responsável por automatizar a geração de grandes volumes de programas C. Esse script executa o Csmith repetidamente utilizando opções como `-max-funcs 2`, `-no-global-variables`, entre outras configurações que restringem a complexidade e asseguram compatibilidade posterior com o processo de conversão para ROBL. O código do script encontra-se disponível no repositório público:

<https://github.com/CalebeMiranda/Pipeline-training/tree/main/AVR/CSMITH>.

Com esse procedimento, foram gerados aproximadamente 80 mil códigos sintéticos em C. Embora a meta inicial fosse atingir 100 mil programas, estimativas de tempo de treinamento do modelo indicaram que 100k exemplos tornariam o processo excessivamente longo. Assim, adotou-se o conjunto de 80k códigos sintéticos como solução de compromisso entre volume, diversidade e viabilidade computacional.

4.2.1 Códigos reais obtidos do AnghaBench

A segunda fonte de dados consiste em códigos reais provenientes do repositório AnghaBench (SILVA et al., 2021), disponível em <https://github.com/brenocfg/AnghaBench/tree/master>. O AnghaBench contém aproximadamente um milhão de funções em C, extraídas automaticamente de diversos projetos hospedados no GitHub, apresentando grande variedade de estilos, padrões de escrita e estruturas de código típicas de aplicações reais.

Como os programas do AnghaBench já são curtos e subdivididos em funções isoladas, não houve necessidade de tratamento ou pré-processamento adicional. Para compor o subconjunto utilizado nesta pesquisa, foram selecionados 10 mil códigos aleatórios a partir dos diretórios do AnghaBench, utilizando o comando `find` no terminal para amostragem simples dos arquivos disponíveis.

Ao final desta etapa, o conjunto inicial de programas em C, combinando cerca de 80k códigos sintéticos e 10k códigos reais, foi consolidado e preparado para as etapas seguintes do pipeline.

4.3 Conversão de C para RobL

A segunda etapa do pipeline consiste exclusivamente na conversão dos programas escritos em C para a linguagem ROBL. Essa conversão é essencial para permitir que os códigos sejam compiláveis pelo Robcmp, e possam ser considerados aplicáveis em uma MCU AVR.

Para realizar a conversão, utilizou-se o conversor `c2rob`, disponível no repositório oficial do projeto Robotics Language: <https://github.com/thborges/c2rob>. O repositório foi clonado, e modificações pontuais foram aplicadas ao `Makefile` e a arquivos auxiliares, a fim de assegurar que o código convertido estivesse configurado especificamente para a arquitetura alvo adotada neste trabalho, o ATmega328P. Essas alterações garantiram conformidade com as características do backend AVR associado ao Robcmp, que compilava os códigos assim que já estivessem traduzidos, garantindo códigos RobL fiéis e compiláveis.

Como a linguagem ROBL não implementa todos os recursos da linguagem C,

tornou-se necessária a aplicação de um mecanismo de filtragem automática dos programas obtidos na etapa anterior. A filtragem foi realizada diretamente a partir da saída do `c2rob`: sempre que o conversor emitia um erro de tradução o código correspondente era automaticamente excluído do fluxo principal. Assim, o erro de sintaxe emitido pelo `c2rob` funcionou como um critério objetivo de incompatibilidade.

Ao final desta fase, obtém-se exclusivamente o conjunto de programas que foram efetivamente convertidos para ROBL e considerados aptos para seguir para as etapas seguintes do pipeline.

O conjunto resultante possui 1000 programas do CSmith e 803 do AnghaBench, totalizando 1803 códigos RobL válidos e compiláveis para a arquitetura AVR. Foi escolhido manter esse volume de códigos, mesmo que menor que o inicialmente previsto, devido principalmente às limitações financeiras e de tempo de treinamento do modelo, que ultrapassavam o período disponível para elaboração deste que este trabalho. Os códigos convertidos do Csmith possuem em média 45 Kb de tamanho, com aproximadamente 1200 linhas de código cada, enquanto os códigos do AnghaBench são menores, com cerca de 2 Kb e 100 linhas em média. Essa diferença de tamanho se deve ao fato que o repositório AnghaBench já disponibiliza códigos curtos e isolados, geralmente funções ou pequenos trechos retirados de códigos maiores, enquanto o Csmith gera códigos mais extensos, mas não necessariamente complexos, dado que são compiláveis mas não possuem uma aplicação prática definida, exceto o teste de compiladores.

4.4 Método exaustivo de busca da melhor sequência de passes de otimização.

A terceira etapa do pipeline consiste na identificação da melhor sequência de passes de otimização para cada código convertido, com o objetivo de minimizar o tamanho do código objeto gerado para a plataforma AVR. Embora a metodologia completa descrita por Faustino et al. (2021) não tenha sido aplicada integralmente, elementos essenciais do seu trabalho foram incorporados ao presente estudo.

A base utilizada nesta etapa foi o conjunto de sequências de otimização conhecido como *Optimization Cache*, disponibilizado pelos autores em seu repositório oficial. Trata-se de um conjunto pré-computado de sequências particularmente eficazes para redução de tamanho de código no LLVM, servindo como uma alternativa robusta às estratégias tradicionais, como a `-Oz`. Contudo, essas sequências foram originalmente elaboradas para a versão LLVM-10, enquanto esta pesquisa adotou a versão mais recente, LLVM-20.

Durante os testes iniciais observou-se que apenas cerca de 30% das sequências podiam ser aplicadas diretamente no LLVM-20. As demais falhavam devido a modificações

internas do compilador e no ambiente LLVM. Para contornar esse problema, realizou-se uma conversão manual dos passes, mapeando os equivalentes entre as versões 10 e 20 do LLVM. Após esse trabalho de ajuste, tornou-se possível recuperar 100% da lista original, composta por **1289 sequências únicas**, agora compatíveis com a infraestrutura moderna do LLVM-20.

Com a lista completa e compatível, foi construído um processo de compilação exaustiva utilizando o script `rodar_seq_uniq_AVR.sh`. Esse script recebe como entrada o arquivo `sequencias_unicas.txt`, contendo todas as 1289 sequências convertidas, e executa o seguinte procedimento para cada arquivo IR:

1. Aplicação dos passes selecionados ao arquivo LLVM-IR do código de entrada;
2. Compilação para assembly AVR;
3. Geração do arquivo objeto correspondente;
4. Extração e armazenamento do DEC (tamanho em bytes do código objeto);
5. Armazenamento dos artefatos referentes à melhor sequência parcial (arquivos `.s`, `.o` e `.ll`).

Ao término da execução do script para cada código, é gerado um arquivo `melhor.txt` contendo a sequência ótima responsável pelo menor tamanho de código objeto produzido.

Como o volume de códigos do dataset é significativo, a execução completa do processo em modo sequencial seria impraticável. Para lidar com essa limitação, desenvolveu-se o script `parallel-run.py`, responsável por paralelizar a execução do `rodar_seq_uniq_AVR.sh` utilizando nove *threads* da CPU. Esse mecanismo distribui simultaneamente diferentes códigos entre os núcleos disponíveis, reduzindo o tempo total de execução.

Por fim, empregou-se o script `coleta_melhores.py` para consolidar os resultados. Esse script percorre todas as pastas geradas durante a fase de otimização e extrai o arquivo `melhor.txt` de cada código analisado, compilando todos esses resultados em um único arquivo `melhores.csv`. Este arquivo constitui o conjunto final de sequências ótimas associado ao dataset e será utilizado nas etapas posteriores de formação dos pares *prompt*/*label* e preparação do conjunto de treinamento do modelo. Todos os códigos e arquivos gerados estão disponíveis no repositório: <https://github.com/CalebeMiranda/Pipeline-training/tree/main/AVR>.

4.5 Geração dos pares *prompt/label* e elaboração do *dataset* de treinamento

Com as etapas anteriores concluídas, a conversão para ROBL, a geração do LLVM-IR não otimizado e a identificação da melhor sequência de passes para cada programa, torna-se possível estruturar o conjunto de dados que será utilizado para o treinamento supervisionado do modelo de linguagem. Esta fase consiste em transformar cada código processado em um par composto por *prompt* e *label*, seguindo o padrão metodológico proposto no trabalho original do LLM-Compiler (CUMMINS et al., 2023).

Após a execução do script `rodar_seq_uniq_AVR.sh`, cada programa possui três informações fundamentais: o arquivo LLVM-IR resultante da compilação sem otimizações expressivas, a melhor sequência de passes encontrada dentre as 1289 sequências testadas e o arquivo de assembly otimizado correspondente a essa sequência. Com esses elementos, torna-se possível construir pares de entrada e saída adequados ao processo de aprendizado supervisionado (*Supervised Fine-Tuning*). Ao total foram gerados 1721 pares *prompt/label*, derivados dos 1803 códigos que foram convertidos para ROBL e passaram pelo processo de otimização. Essa diferença se dá ao fato de que alguns códigos não geraram sequências válidas ou apresentaram falhas durante a compilação e portanto são descartados do conjunto de formação de pares. As falhas durante a compilação se dá devido a conversões realizadas pelo c2rob que ainda não são suportadas pela ROBL.

Em cada instância do conjunto de dados, o *prompt* consiste em uma instrução textual em inglês seguida do código LLVM-IR não otimizado. A instrução foi definida conforme o padrão empregado por (CUMMINS et al., 2023), mantendo a estrutura funcional do modelo original. A mensagem inicial é:

> *"Tell me what passes to run on the following LLVM-IR to reduce the object file size for avr assembly."*

Em seguida, o código LLVM-IR correspondente ao programa é anexado diretamente após essa instrução, compondo o conteúdo completo do *prompt*. Esse formato padronizado orienta o modelo a identificar a tarefa de seleção de passes como um problema de recomendação condicional baseado no IR de entrada.

O *label*, por sua vez, contém a resposta correta esperada para o exemplo. Ele segue o formato descritivo adotado no LLM-Compiler, sendo estruturado da seguinte forma:

> *"Run the following passes <lista_de_passes> to reduce the object file size to <tamanho_em_bytes>."*

Após essa mensagem inicial, inclui-se o código assembly gerado a partir da aplicação da sequência ótima de passes. O *label* engloba, tanto a justificativa textual quanto o resultado concreto da aplicação dos passes, replicando, parcialmente, a estrutura esperada pelo treinamento do modelo original.

A geração automática desses pares foi realizada por meio do script `promptLabel.py`, disponível no repositório público do pipeline (<https://github.com/CalebeMiranda/Pipeline-training/tree/main/AVR>). O script percorre todos os diretórios contendo os arquivos LLVM-IR, as sequências ótimas (`melhor.txt`) e os arquivos de assembly otimizados, combinando essas informações e produzindo um arquivo final no formato `JSONL`. Cada linha do arquivo está mapeada por uma coluna "messages", seguido de "content", assim como esperado pelo serviço de *autotrain* do *hugging face*, e a linha representa um par de dados, contendo os campos:

- `"prompt"`: instrução textual seguida do LLVM-IR não otimizado;
- `"label"`: sequência ótima de passes, redução de tamanho e assembly correspondente;

Esse procedimento resulta em um dataset estruturado, padronizado e diretamente compatível com a etapa de *fine-tuning* do modelo LLM-Compiler, garantindo consistência com a metodologia original e adequação ao domínio específico da arquitetura AVR.

4.6 Configuração de treinamento do modelo

A etapa final do pipeline consiste na configuração do ambiente necessário para o refinamento do modelo LLM-Compiler (CUMMINS et al., 2023), de modo a adaptá-lo ao domínio específico de otimização de código para microcontroladores AVR. Diferentemente do trabalho original, cujo treinamento foi conduzido em infraestrutura própria e com forte customização de hiperparâmetros, o presente trabalho utiliza recursos em nuvem da plataforma Hugging Face, adequados às limitações de escopo e hardware disponíveis.

Para esse fim, foi criado um ambiente dedicado na plataforma, acessível por meio do espaço público[1]. Esse ambiente foi configurado para possibilitar o *Supervised Fine-Tuning* (SFT) do modelo, utilizando uma GPU Nvidia A10G na configuração *large*, contendo 12 vCPUs, 46 GB de RAM e 24 GB de VRAM. A opção pela execução em nuvem se justifica pela necessidade de memória de vídeo compatível com janelas de contexto amplas, inviáveis no hardware local disponível para experimentos.

O processo de treinamento foi conduzido utilizando a ferramenta AutoTrain Advanced [2], que possibilita a realização de ajuste fino supervisionado sem a necessidade de

[1] <https://huggingface.co/spaces/Cal-mfbc5446/STF-PFC2>
[2] <https://huggingface.co/docs/autotrain>

desenvolvimento de código adicional, operando por meio de interface gráfica. Embora essa abordagem reduza a flexibilidade na configuração de hiperparâmetros, permite um fluxo de experimentação mais rápido e simplificado, adequado ao escopo deste trabalho.

Os parâmetros de treinamento empregados foram adaptados a partir das configurações originalmente utilizadas por Cummins et al. (2023). Entretanto, apenas parte das estratégias apresentadas no trabalho original pôde ser reproduzida neste ambiente. Em particular, foi possível utilizar:

- O *scheduler* do tipo Cosine, também presente no LLM-Compiler;
- O otimizador AdamW, sem a possibilidade de especificar manualmente os valores de β_1 e β_2;
- *weight decay* e taxa de aprendizado configuráveis dentro dos limites da ferramenta.

Além disso, devido às restrições de memória da GPU disponibilizada na plataforma, o parâmetro `model_max_length` teve de ser ajustado para 8 192 tokens, metade do utilizado no trabalho original (16 384 tokens). A redução foi necessária para garantir que o modelo pudesse ser carregado e treinado sem exceder a capacidade de VRAM.

Todos os hiperparâmetros utilizados no treinamento foram registrados no arquivo `parametros-de-treinamento-IA.txt`, disponível no repositório oficial do pipeline neste GitHub. O arquivo documenta as configurações efetivamente aplicadas, incluindo divisões do conjunto de dados, *learning rate*, tamanho do lote e número de épocas planejado.

Para a composição do conjunto de treinamento, seguiu-se a divisão tradicionalmente adotada em tarefas de *fine-tuning* supervisionado: 80% dos pares *prompt*/*label* foram destinados ao treinamento, 10% ao conjunto de validação e 10% ao conjunto de teste. A validação permite monitorar sobreajuste, enquanto o conjunto de teste é reservado para a avaliação final do modelo refinado durante a etapa de resultados.

Embora o treinamento completo não tenha sido executado dentro do prazo deste trabalho, toda a infraestrutura necessária para sua realização foi preparada, incluindo a configuração do ambiente computacional, a definição dos hiperparâmetros, a organização do dataset final e a preparação do pipeline para execução direta do SFT.

5 Resultados

A partir do pipeline implementado neste trabalho foi possível construir um conjunto de 1721 pares *prompt*/*label* destinado ao treinamento supervisionado do LLM-Compiler especializado para AVR. Cada *prompt* corresponde ao código em ROBL compilado para LLVM-IR (versão não otimizada, compilada com a flag `-O1`), e o respectivo *label*, contém a melhor sequência de passes de otimização identificada para a plataforma alvo (ATmega328P) e o código assembly gerado após a aplicação desses passes. O conjunto foi produzido a partir de aproximadamente 90 mil programas originalmente escritos em C, sendo estes, 80 mil gerados por Csmith e 10 mil coletados no AnghaBench. Dos quais, arbitrariamente 1000 códigos do Csmith e 803 códigos do AnghaBench, foram convertidos para ROBL, totalizando um total de 1803 códigos compilados com diferentes sequências de passes e avaliados quanto ao tamanho do objeto gerado para a plataforma AVR.

A seleção das melhores sequências para cada programa foi realizada utilizando a lista de sequências derivada do trabalho de Faustino et al. (2021), denominada de *Optimization Cache* em conjunto com a comparação contra flags tradicionais do LLVM (`-Oz`, `-Os`, `-O1`, `-O2`, `-O3`). Essa etapa foi automatizada pelos scripts desenvolvidos no repositório do pipeline (por exemplo, `rodar_seq_uniq_AVR.sh`, `parallel-run.py` e `coleta_melhores.py`), que percorrem o conjunto de 1.289 sequências únicas consideradas e registram, para cada programa, a sequência que produz o menor tamanho de objeto e o respectivo `.s` / `.o` / `.ll` resultante. A comparação detalhada entre as sequências apresentadas por Faustino et al. (2021) e as flags padrões encontra-se documentada na planilha *Comparação Otimização Faustino vs Padrão* disponível neste link.

Dessa forma, os resultados práticos deste trabalho incluem os scripts:

- O script *rodar_seq_uniq_AVR.sh* responsável por aplicar, para cada programa em LLVM-IR, todas as 1289 sequências de otimização únicas derivadas do trabalho de Faustino et al. (2021). Compilando o código para AVR para cada sequência, armazenando os arquivos resultantes (.s, .o, .ll) e registrando, para cada programa, aquela sequência que produz o menor tamanho de objeto, definindo assim o "melhor caso".

- O script *parallel-run.py* que atua paralelizando a execução do *rodar_seq_uniq_AVR.sh* em múltiplos núcleos da CPU, permitindo processar simultaneamente diversos códigos. Reduzindo o tempo total de execução.

- O script *coleta_melhores.py* que percorre todas as subpastas de resultados produzidas pelas compilações e extrai, para cada programa, o arquivo objeto melhor.o e seu

tamanho em bytes. Esses valores são consolidados em um arquivo único *melhores.csv*, organizado como uma tabela contendo o nome do programa e o tamanho pós otimização.

- O script *promptLavel.py* que realiza a montagem dos pares de treinamento. Carregando tanto o IR não otimizado quanto o assembly otimizado, além da sequência de passes ótima e do tamanho final do objeto, e organiza essas informações no formato JSONL adotado pelo LLM-Compiler.

Além dos scripts, este trabalho também contribuiu com dados prontos para consumo, que é o caso dos arquivos:

- **`dec_otimizacoes_GLOBAL.csv`**, contendo o registro global dos resultados das compilações realizadas pelas flags padrões. Esse arquivo sintetiza, para cada programa e para cada flag, o tamanho do objeto gerado e permite validação e comparação de desempenho entre diferentes tecnicas de otimização.

- **`melhores.csv`**, que consolida somente as melhores sequências descobertas para cada programa após a execução da metodologia exaustiva. Esse arquivo é fundamental para a etapa de construção dos pares *prompt*/*label*, pois fornece a sequência de passes selecionada como rótulo supervisionado do modelo.

- **`Dataset-AVR-Completo.jsonl`**, que contém o dataset final que poderá ser utilizado no treinamento supervisionado. Cada linha contém um objeto JSON estruturado com o *prompt* (texto inicial + código LLVM-IR não otimizado) e o *label* (sequência ótima de passes + assembly otimizado). Esse arquivo segue o formato recomendado para pipelines do Hugging Face e serve como dataset de refinamento do LLM-COMPILER para AVR.

- **`requirements.txt`**, que especifica o ambiente mínimo necessário para executar os scripts do pipeline e manipular os dados gerados, garantindo reprodutibilidade e facilitando a replicação do experimento em outras máquinas ou em ambientes.

Adicionalmente, este trabalho também contribuiu no processo de configuraçao do ambiente de refinamento do modelo, ao adicionar o espaço público `Cal-mfbc5446/STF-PFC2`[1] na plataforma Hugging Face, disponibilizar o dataset de treinamento já convertido no formato de jsonl e parquet (formato que o hugging face armazena em nuvem o dataset)[2], e também disponibilizando todas as configurações de hiperparâmetros que podem ser utilizadas no refinamento do modelo, no arquivo `parametros-de-treinamento-IA.txt`[3].

[1] Disponível em <https://huggingface.co/spaces/Cal-mfbc5446/STF-PFC2>
[2] disponível em <https://huggingface.co/datasets/Cal-mfbc5446/Dataset-AVR-Completo>
[3] Disponível em <https://github.com/CalebeMiranda/Pipeline-training/tree/main/AVR>

Outro resultado deste trabalho é a análise comparativa entre as sequências do Faustino e as flags padrão, presente na Figura 6, que mostra uma vantagem significativa das sequências especializadas em relação à flag padrão `-Oz`, com *52,1%* dos 1803 casos apresentandos demonstrando redução de tamanho de código em relação à `-Oz`, *47,9%* dos casos apresentando resultados equivalentes e apenas *0,1%* dos casos apresentando piora em relação à `-Oz`. Este resultado, corrobora com a relevância de usar conjuntos de sequências especializados como ponto de partida para a construção de pares de treinamento e é disponibilizada como material de apoio ao presente trabalho.

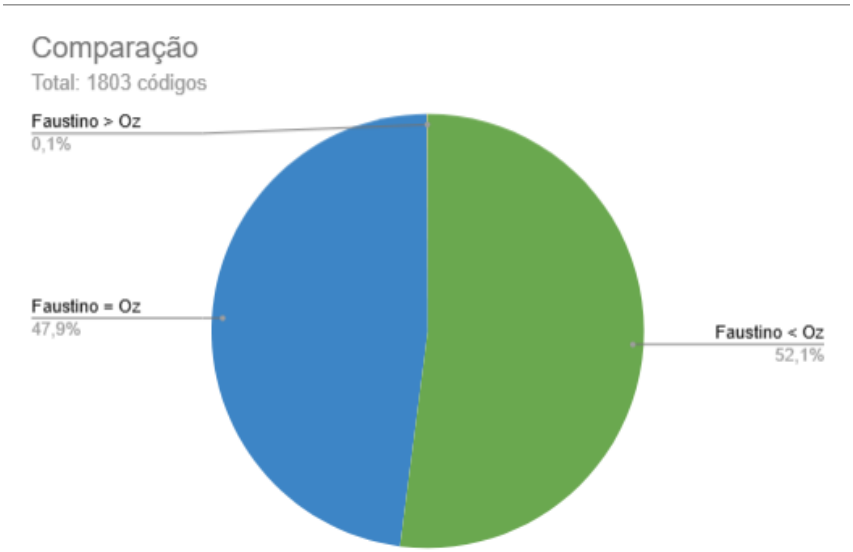

Figura 6 – Comparação de Otimização. Fonte: <https://github.com/CalebeMiranda/Pipeline-training/tree/main/AVR>(Elaboração Própria).

Vale a pena mencionar, que esta pesquisa, elaborada em conjunto com outro trabalho de tema semelhante, obteve um modelo refinado para a família de MCUs Stm32[4], cujo pipeline de geração de dados e configuração de treinamento seguiu uma metodologia semelhante à descrita neste trabalho. O modelo refinado é capaz de sugerir sequências de passes compatíveis com aquelas observadas no treinamento, porém após realizado testes de inferência o modelo apresentou um comportamento de sucesso esporádico, retornando padrões de otimização previamente observados somente para alguns códigos provenientes do conjunto de treinamento do AnghaBench. Portanto, esse comportamento inconsistente não é suficiente para considerar esse resultado como conclusivo, porém é um indicativo positivo de que o método empregado em ambos trabalhos é viável e pode ser aprimorado em trabalhos futuros, visto que o tamanho reduzido do conjunto de treinamento e as limitações de tempo, também foram os principais problemas encontrados no refinamento deste modelo de IA.

[4] Disponível em <https://huggingface.co/Cal-mfbc5446/LlmCompiler-Stm32FineTunningFinal>

Portanto, o principal resultado deste trabalho é a implementação de um pipeline reprodutível e público que cobre todo o processo:

- Geração e seleção de programas em C
- Conversão automatizada para ROBL
- Compilação com listas exaustivas de passes (1.289 sequências únicas) e comparação com flags padrão
- Coleta das melhores sequências e montagem dos pares *prompt*/*label*
- Preparação do ambiente de fine-tuning

6 Discussão

Durante o desenvolvimento deste trabalho, observou-se uma série de limitações que impactaram diretamente o escopo e os resultados alcançados. Os principais desafios enfrentados foram decorrentes de restrições de tempo, orçamento e gastos computacionais. Estes problemas acarretaram em uma série de mudanças no planejamento inicial, que influenciaram diretamente o escopo deste trabalho. Diversas mudanças foram aplicadas no decorrer desta pesquisa, porém, a mais significativa delas foi que o refinamento e consequentemente a inferência e validação, do modelo LLM-Compiler para AVR não poderia ser concluído dentro do prazo estipulado.

Outros problemas relevantes que causaram uma mudança no escopo deste projeto foi a estimativa de tempo e recursos associados à execução do pipeline completo de autotunnig proposto por Faustino et al. (2021). A execução do método completo demandaria tempo e adicionaria complexidade a um projeto já limitado por prazos e recursos. Em função dessas limitações, optou-se por restringir o autotuning a uma lista predefinida de sequências a *Optimization Cache* derivada de outros trabalhos de Faustino et al. (2021) e por processar um subconjunto controlado de programas priorizando a reprodutibilidade do pipeline, em detrimento da exploração exaustiva do espaço combinatório de passes. Além disso, por causa destas restrições de tempo e capacidade computacional, o volume de programas processados foi reduzido de 90 mil códigos iniciais para aproximadamente 1.800 códigos efetivamente convertidos para ROBL e avaliados com a lista de 1.289 sequências únicas, o que por sua vez impactou diretamente no tamanho do dataset final produzido para o treinamento do modelo, que ficou reduzido para aproximadamente 1.700 pares *prompt/label*.

Outra limitação relevante refere-se ao tamanho dos pares *prompt/label* em termos do número de tokens comparado à janela de contexto do modelo original e a memória disponível para o fine-tuning. Em razão da GPU utilizada (Nvidia A10G alugada via Hugging Face, que possui 12vCPU, 46 GB de RAM e 24 GB de VRAM), o parâmetro `model_max_length` foi ajustado para 8192 tokens. Vários exemplos gerados pelo pipeline excederam esse limite, exigindo truncamento para viabilizar o treinamento. O truncamento potencialmente remove contexto semântico importante para a escolha adequada de passes, prejudicando a qualidade do aprendizado. Porém, foi escolhido preservar os prompts completos no repositório, pensando em infraestruturas futuras que permitam janelas de contexto maiores.

Outros problemas técnicos superados foram a conversão e mapeamento manual dos passes da lista *Optimization Cache*, originalmente definidos para LLVM-10, para torná-los utilizáveis no LLVM-20. Essa etapa de adaptação introduziu um esforço adicional para

garantir que todas as sequências pudessem ser aplicadas corretamente, evitando falhas de compilação e garantindo a integridade do processo de autotuning.

7 Conclusão e Trabalhos Futuros

Neste trabalho foi desenvolvido um pipeline reprodutível para a geração de dados de treinamento visando o refinamento do modelo LLM-Compiler na tarefa de otimização de código em microcontroladores AVR (em específico para ATmega328P). O processo abrangeu automação das etapas de geração de código C, conversão automática para ROBL (via c2rob), compilação para LLVM-IR com o backend AVR, aplicação de 1.289 sequências de passes de otimização (baseadas em (FAUSTINO et al., 2021)) e comparação com flags tradicionais do LLVM, gerando a construção de pares de prompt e label para fine-tuning. A geração de diversos exemplos de treinamento aplicando sequências de passes de otimização está alinhada com estratégias usadas em LLMs especializados para compiladores, evidenciando que é possível obter, de forma automatizada, o conjunto de dados necessário para ajustar o modelo.

Como resultado, foi produzido um conjunto contendo cerca de 1.721 pares (prompt/label) dentre os 1.803 códigos compilados. Embora de tamanho limitado, esse conjunto foi gerado de forma consistente e reprodutível, segundo a metodologia proposta. As principais limitações observadas incluem o elevado custo computacional necessário para executar as milhares de compilizações otimizadas e o tempo de execução deste projeto. Essas restrições impediram a execução do refinamento final do modelo LLM-Compiler para AVR dentro deste trabalho. Entretanto, estudos similares voltados a microcontroladores (por exemplo, arquiteturas ARM Cortex-M como STM32) já ilustram a viabilidade prática dessa metodologia, servindo de referência para outros trabalhos que seguem a mesma linha de pesquisa.

Apesar das limitações enfrentadas, o pipeline desenvolvido e o conjunto de dados produzidos configuram uma base sólida para pesquisas futuras. A estrutura proposta mostra, na prática, que é possível automatizar de ponta a ponta o processo de preparação de dados para o refinamento de modelos de linguagem voltados à otimização de código embarcado. Embora o presente trabalho não tenha realizado o ajuste fino do modelo, os recursos disponibilizados estabelecem as condições necessárias para que essa etapa seja facilmente executada em estudos posteriores.

7.1 Trabalhos Futuros

Para aprofundar e ampliar os resultados deste estudo, sugerem-se as seguintes direções de pesquisa:

- Ampliar o tamanho do dataset de treinamento, expandindo as conversões disponíveis

no c2rob ou a medida que novos recursos forem implementados na ROBL.

- Fine-tuning para AVR: realizar efetivamente o refinamento (fine-tuning) do modelo LLM-Compiler para a arquitetura AVR usando os dados já produzidos, possibilitando validar na prática o ganho de qualidade de código otimizado em compilação real.
- Expansão para outras variantes AVR: estender o pipeline para suportar diferentes variantes da arquitetura AVR (além do ATmega328P), de modo a coletar e gerar dados de treinamento específicos para outros microcontroladores da família, aumentando a abrangência do modelo.
- Geração automática de sequências de passes: explorar estratégias de geração automática de novas sequências de passes de otimização, indo além do conjunto fixo utilizado (*Optimization Cache*). Por exemplo, empregar amostragem aleatória ou algoritmos genéticos para cobrir um espaço maior de combinações de passes e enriquecer o conjunto de dados.
- Integração no robcmp: incorporar o modelo refinado ao compilador robcmp por meio de uma nova opção (por exemplo, uma flag -IA), permitindo que o compilador realize inferência com o modelo LLM durante o processo de compilação. Essa integração viabilizaria a aplicação direta do modelo em tempo de compilação para escolher otimizações, aproximando a pesquisa de um uso prático.

Essas iniciativas poderão superar as limitações atuais e aproximar o pipeline de uma aplicação prática, ampliando o potencial de otimização de código para sistemas embarcados e fundamentando pesquisas futuras em otimização de compiladores.

Referências

AHO, A. V.; SETHI, R.; ULLMAN, J. D. Compiladores: Princıpios, técnicas e ferramentas. *LTC, Rio de Janeiro, Brasil*, p. 219–276, 2007. Citado 3 vezes nas páginas 16, 19 e 20.

Atmel Corporation. *ATmega328P: 8-bit AVR Microcontroller with 32K Bytes In-System Programmable Flash*. [S.l.], 2015. Datasheet, 7810D–AVR–01/15. Disponível em: <https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcontrollers-ATmega328P_Datasheet.pdf>. Citado 2 vezes nas páginas 13 e 22.

BODEN, M. A. Inteligencia artificial. Turner, 2017. Citado na página 25.

CHANG, Y. et al. A survey on evaluation of large language models. *ACM transactions on intelligent systems and technology*, ACM New York, NY, v. 15, n. 3, p. 1–45, 2024. Citado na página 28.

COOPER, K.; TORCZON, L. *Engineering a Compiler*. Morgan Kaufmann, 2012. (Morgan Kaufmann). ISBN 9780120884780. Disponível em: <https://books.google.com.br/books?id=CGTOlAEACAAJ>. Citado 10 vezes nas páginas 15, 16, 17, 18, 19, 20, 21, 22, 28 e 29.

COSTA, R. H. P. et al. *Compiladores*. [S.l.]: Editora Científica, 2023. ISBN 978-65-00-68043-0. Citado na página 20.

CUMMINS, C. et al. Large language models for compiler optimization. *arXiv preprint arXiv:2309.07062*, 2023. Citado 9 vezes nas páginas 11, 12, 13, 20, 29, 31, 37, 38 e 39.

FAUSTINO, A. et al. New optimization sequences for code-size reduction for the llvm compilation infrastructure. In: *Proceedings of the 25th Brazilian Symposium on Programming Languages*. [S.l.: s.n.], 2021. p. 33–40. Citado 5 vezes nas páginas 32, 35, 40, 44 e 46.

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 8 vezes nas páginas 15, 17, 18, 19, 21, 22, 28 e 29.

HUSSAIN, A. et al. Programming a microcontroller. *Int. J. Comput. Appl*, v. 155, n. 5, p. 21–26, 2016. Citado 2 vezes nas páginas 12 e 22.

Intel. *Processador Intel® Core™ i9-10900K (20M Cache, até 5,30 GHz) Especificações*. Disponível em: <https://www.intel.com.br/content/www/br/pt/products/sku/199332/intel-core-i910900k-processor-20m-cache-up-to-5-30-ghz/specifications.html>. Citado na página 13.

LATTNER, C.; ADVE, V. Llvm: A compilation framework for lifelong program analysis & transformation. In: IEEE COMPUTER SOCIETY. *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO)*. [S.l.], 2004. p. 75–86. Citado na página 29.

LI, Q.; YAO, C. *Real-Time Concepts for Embedded Systems*. Boca Raton, London, New York: CRC Press, Taylor & Francis Group, 2003. Citado na página 12.

LUDERMIR, T. B. Inteligência artificial e aprendizado de máquina: estado atual e tendências. *Estudos Avançados*, SciELO Brasil, v. 35, p. 85–94, 2021. Citado 2 vezes nas páginas 25 e 26.

MERNIK, M.; HEERING, J.; SLOANE, A. M. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*, ACM New York, NY, USA, v. 37, n. 4, p. 316–344, 2005. Citado na página 23.

MORANDÍN-AHUERMA, F. ¿ what is artificial intelligence? 2022. Citado na página 25.

MUHAMMAD, I.; YAN, Z. Supervised machine learning approaches: A survey. *ICTACT Journal on Soft Computing*, v. 5, n. 3, 2015. Citado 2 vezes nas páginas 25 e 26.

OLIVEIRA, T. B. *Robcmp: Compilador para Microcontroladores voltado à Robótica e IoT*. 2024. <https://github.com/thborges/robcmp>. Acessado em maio de 2025. Citado 2 vezes nas páginas 13 e 24.

PURINI, S.; JAIN, L. Finding good optimization sequences covering program space. *ACM Trans. Archit. Code Optim.*, Association for Computing Machinery, New York, NY, USA, v. 9, n. 4, p. 56:1–56:23, 2013. ISSN 1544-3566. Disponível em: <https://doi.org/10.1145/2400682.2400715>. Citado 2 vezes nas páginas 31 e 32.

RUSSELL, S.; RUSSELL, S.; 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 5 vezes nas páginas 24, 25, 26, 27 e 29.

SILVA, A. F. D. et al. Anghabench: A suite with one million compilable c benchmarks for code-size reduction. In: IEEE. *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. [S.l.], 2021. p. 378–390. Citado na página 34.

The LLVM Project. *LLVM Language Reference Manual*. 2025. <https://llvm.org/docs/LangRef.html>. Acessado em 18 de maio 2025. Citado na página 20.

VASWANI, A. et al. Attention is all you need. *Advances in neural information processing systems*, v. 30, 2017. Citado na página 28.

WANG, Z.; O'BOYLE, M. Machine learning in compiler optimisation. *arXiv preprint arXiv:1805.03441*, 2018. Citado 2 vezes nas páginas 12 e 29.

WHITE, E. *Making Embedded Systems: Design Patterns for Great Software*. [S.l.]: "O'Reilly Media, Inc.", 2024. Citado 3 vezes nas páginas 22, 23 e 24.

YANG, X. et al. Finding and understanding bugs in c compilers. In: *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. [S.l.: s.n.], 2011. p. 283–294. Citado na página 33.

ZHAO, H. et al. Explainability for large language models: A survey. *ACM Transactions on Intelligent Systems and Technology*, ACM New York, NY, v. 15, n. 2, p. 1–38, 2024. Citado na página 28.