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

Trabalho de Conclusão de Curso apresentado ao curso de Bacharelado em Ciências da Computação, como requisito para obtenção do grau final na disciplina de Trabalho de Conclusão de Curso 1.

Universidade Federal de Jataí

Orientador: Prof. Dr. Thiago Borges de Oliveira

Jataí-GO

2025

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

Introdução

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 rápida de execução de código, respectivamente, porém, também aumentam o tamanho do código binário produzido e tornam o processo de compilação proporcionalmente mais

[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>

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

Advanced RISC de 8 bits, foco de atuação deste trabalho, presente em microcontroladores como o ATmega2560, possuem características extremamente limitadas como espaço de memória *flash* de 256 KB, SRAM interna de 8 KB, um *clock* de 16 MHz (Atmel Corporation, 2005). 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.

Diante desse cenário, o presente trabalho tem como objetivo refinar o modelo LLM-Compiler (CUMMINS et al., 2023), aplicando nele um treinamento adicional com programas e otimizações no domínio de microcontroladores, especificamente a plataforma alvo AVR do LLVM.

Nossa proposta envolve a formação de uma base de dados específica, composta por códigos na linguagem ROBL, previamente convertidos da linguagem C, e suas respectivas listas de passes ótimos extraídas por uma busca semi-exaustiva. Com essa base, pretende-se realizar o *fine-tuning* do LLM-Compiler original, fornecendo os códigos IR da base de dados em ROBL, já compilados pelo Robcmp. 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 são:

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 e encontrar os passes ótimos de instruções para cada código.

3. Realizar o *fine-tuning* do modelo LLM-Compiler com essa base adaptada;

4. Avaliar a performance do modelo ajustado na geração de listas de passes para novos códigos não vistos durante o treinamento;

5. Comparar os resultados obtidos com as estratégias tradicionais de compilação, como -Oz, verificando ganhos em tamanho de código, contagem de instruções e desempenho.

1 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.

1.1 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.

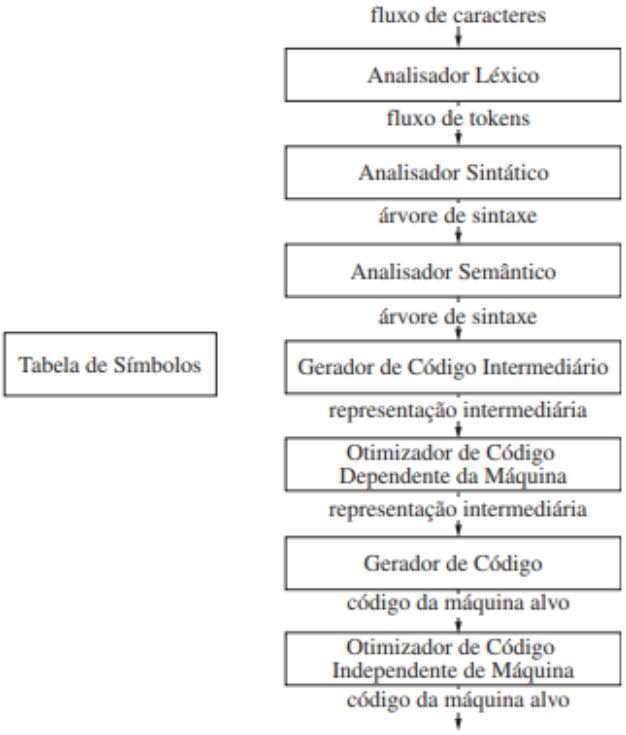

Figura 1 – Fases de um Compilador. Fonte: (AHO; SETHI; ULLMAN, 2007)

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.

1.1.1 *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).

1.1.2 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).

1.1.3 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.

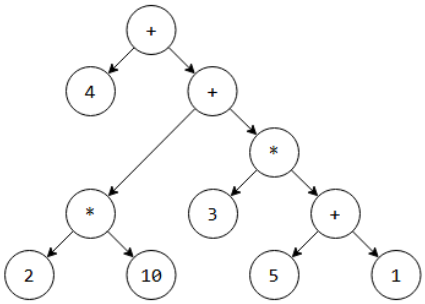

Figura 2 – Exemplo de uma *Abstract Syntax Tree* para $4 + 2 * 10 + 3 * (5 + 1)$. Fonte: <https://keleshev.com/abstract-syntax-tree-an-example-in-c/>

1.1.4 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).

1.1.5 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.

1.1.6 *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.

1.1.7 *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.

1.2 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 Atmega2560-16U 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, 2005).

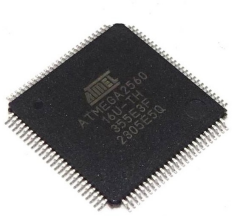

Figura 4 – Foto de exemplo do microcontrolador estudado, Atmega2560-16U. Fonte: <https://www.lojaorielec.com.br/semicondutores/circuito-integrado/microcontrolador/circuito-integrado-microcontrolador-8-bit-atmega164pv-10au-atmel>

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).

1.2.1 *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.

1.2.2 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.

1.3 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).

1.3.1 *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.

1.3.2 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.

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.

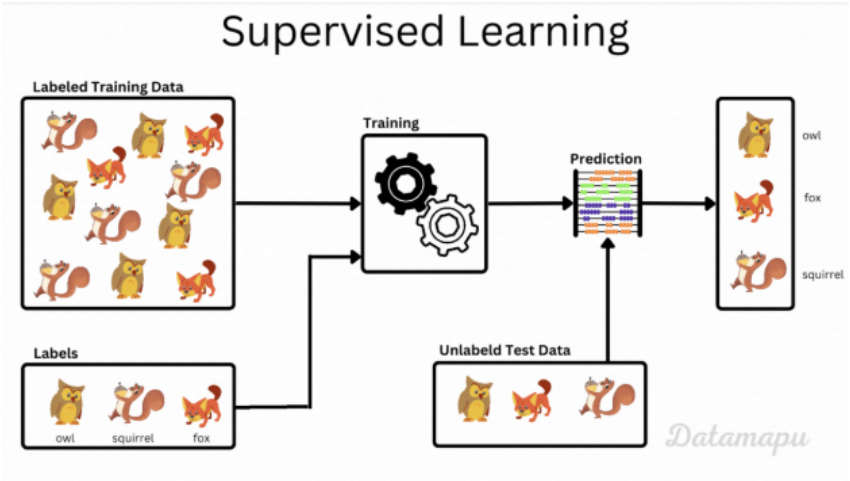

Figura 5 – Representação didática do funcionamento de aprendizagem supervisionado. Fonte: <https://datamapu.com/posts/ml_concepts/supervised_unsupervised>

1.3.3 *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.

1.4 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).

[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/>

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 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.

2 Trabalhos Relacionados

2.1 Introdução

Este capítulo aborda alguns trabalhos relacionados ao refinamento e integração do modelo LLM-Compiler para otimização de código alvo para microcontroladores pertencentes a família AVR, na Robotics Language. Primeiramente será analizado os principais critérios de seleção para estes trabalhos que abordam áreas diretamente ligadas ao escopo desta pesquisa, posteriormente será apresentado um breve resumo sobre cada trabalho e suas contribuições.

2.2 Metodologia de Análise

Para a anáilse dos trabalhos relacionados, foram considerados

2.3 Trabalhos analisados

Considerando os critérios anteriores, foram analisados obras com maior relevância sobre o tema, que abordam de forma direta ou indiretamente o processo de treinamento/refinamento do modelo de IA, geração de datasets e otimização da compilação de programas, em específico redução de tamanho de código. Destaca-se os 3 artigos a seguir que foram selecionados conforme sua prioridade de contribuições para a presente pesquisa.

2.3.1 LLM-COMPILER

O trabalho de (CUMMINS et al., 2023)

2.3.2 Good Optimization Sequences Covering Program Space

O trabalho de (PURINI; JAIN, 2013)

2.3.3 Optimization Sequences for Code-Size Reduction

O trabalho de (FAUSTINO et al., 2021)

3 Metodologia

3.1 Classificação da Pesquisa

O presente trabalho se classifica como uma pesquisa aplicada, no contexto de sua natureza, dado que busca solucionar um problema prático, e o conhecimento gerado é de utilidade imediata sobre o âmbito do assunto abordado. Quanto aos objetivos, é de cunho exploratório, uma vez que investiga o potencial de modelos de linguagem de grande porte (LLMs) na otimização de código para sistemas embarcados.

No que tange aos procedimentos, trata-se de uma pesquisa experimental, já que a análise dos resultados se dá graças ao processo de construção e adaptação de um experimento em um ambiente controlado. Além disso, a abordagem adotada é quantitativa, pois se trata de um processo de avaliação comparativa da performance de um modelo pós *fine-tunning* em relação ao número de instruções presentes e tamanho do código gerado. Por fim, o presente trabalho é classificado como bibliográfico no que tange às fontes, pois depende de artigos e documentos já publicados e presentes no meio acadêmico.

3.2 Abordagem Geral

A metodologia de pesquisa utilizada neste presente trabalho deve ser descrita como uma avaliação de performance de um modelo de LLM já existente, após o processo de *fine-tunning*. Para isso, a construção deste estudo é constituída de etapas de preparação e execução, na seguinte ordem:

1. Elaboração de um *Dataset* (Conjunto de dados) a partir do uso de um software de conversão entre as linguagens de programação C para ROBL;
2. Elaboração de um conjunto de pares de entrada e saída *prompt* e *label* para servirem como material de treinamento e validação do modelo de IA; e
3. Validação do modelo treinado, comparando o resultado de otimização obtido com a integração do modelo de IA no compilador com o resultado obtido através de estratégias convencionais de otimização.

3.3 *Dataset* (Conjunto de dados)

O conjunto de dados utilizado nesta pesquisa será fruto da combinação de códigos feitos na linguagem C de duas fontes diferentes: códigos sintéticos e códigos provenientes

de aplicações reais.

Os códigos de origem sintética serão gerados através do programa Csmith (YANG et al., 2011), um software desenvolvido especificamente para gerar de forma aleatória códigos em C, com o objetivo de testar diversas funcionalidades de novos compiladores, através de um processo chamado *stress testing* (teste por estresse). Serão empregados comandos específicos para limitar a geração destes códigos aleatórios, de forma que contemplem somente os recursos possíveis de serem convertidos para ROBL.

Os códigos de origem de aplicações reais serão obtidos do conjunto de dados chamado AnghaBench (SILVA et al., 2021), uma base de dados composta por cerca de um milhão de funções em C, extraídas de forma automática de diversos repositórios públicos disponíveis na plataforma *GitHub*.

3.4 Conversor C para Rob: `c2rob`

Para que o modelo de IA seja treinado com programas intermediários para MCUs, os códigos provenientes do *dataset*, escritos na linguagem C, serão convertidos para a ROBL. Um conversor, chamado `c2rob`[1], elaborado no âmbito do projeto de pesquisa da *Robotics Language*, será usado para converter o código fonte em C para a linguagem Robcmp, preservando a lógica e o funcionamento do programa original. O conversor foi criado utilizando as ferramentas Flex[2] (versão 2.6.4) e Bison[3] (versão 3.8.2), selecionadas devido a facilidade da geração de analisadores léxicos e sintáticos personalizados. Neste processo, para cada código de entrada em C o `c2rob` gera um código compilável equivalente na gramática do Robcmp, assegurando-se de que todos os códigos convertidos estejam dentro dos limites de aceitabilidade e funcionamento da arquitetura AVR.

Por ser uma ferramenta recém-criada, incluímos no cronograma uma atividade para validar a conversão efetuada pelo `c2rob`. Ainda, como alguns recursos da linguagem C não estão presentes na ROBL, uma etapa adicional será empregada para selecionar códigos possíveis de conversão. A própria falha de tradução, apresentada pelo c2rob como erro de sintaxe, será um indicador de incompatibilidade. Nestes casos, o programa será removido do dataset final.

A priori, esperamos atingir um conjunto de dados mínimo de 100.000 códigos escritos em ROBL, após a conversão. O objetivo é montar um *dataset* abrangente o suficiente para permitir o processo de aprendizado do modelo de inteligência artificial sem acarretar em um tempo de treinamento muito grande, como foi o caso do treinamento original do LLM-Compiler – que durou cerca de 620 dias de GPU. Adicionalmente, o

[1] Disponível em <http://github.com/thborges/c2rob>
[2] https://www.gnu.org/software/flex
[3] https://www.gnu.org/software/bison/

conjunto de dados será composto também, por listas de passes de otimização particulares a cada código, obtidos de forma automática, seguindo o método descrito nas seções à seguir.

3.5 Conjunto de pares prompt e label

Após feita a conversão dos códigos C para ROBL, os dados serão estruturados em pares compostos por uma dupla de código de entrada e a saída, de forma a viabilizar o treinamento supervisionado do modelo de linguagem. Em cada par, o elemento de entrada, *prompt*, corresponde ao código original compilado para LLVM IR, enquanto o elemento de saída, *label* refere-se à sequência ótima de passes de otimização para cada código específico, seguida do código em assembly gerado para a plataforma alvo. A sequência de passes ótima será realizada por meio de um processo de *autotuning*, conforme descrito em (FAUSTINO et al., 2021), que consiste numa busca semi-exaustiva da melhor sequência de passes.

O método sistemático, desenvolvido por Faustino et al. (2021), para encontrar boas sequências de otimização para redução de tamanho de código para LLVM, pode ser dividido nas seguintes etapas: primeiro, um algoritmo genético é utilizado para gerar uma grande seleção de sequências de otimização; os pesquisadores empregaram especificamente a ferramenta YaCoS[4] (*Yet another Compiler Optimization Search*), para um cálculo eficiente das possíveis soluções. Esses passes são testados em múltiplos programas, e os que demonstram os melhores resultados na redução do tamanho do código são selecionados. Posteriormente, um algoritmo de redução é aplicado para remover otimizações desnecessárias das sequências eficazes, proposto por (PURINI; JAIN, 2013). Ele opera de forma iterativa, removendo passes da sequência um por um enquanto a métrica de otimização não é prejudicada, garantindo que apenas os passos essenciais permaneçam, e por final, sequências de passes duplicadas são eliminadas.

3.6 Treinamento

Com o conjunto de *prompt* e *label* elaborado, a próxima etapa consiste em refinar o treinamento do modelo LLM-Compiler (CUMMINS et al., 2023), de forma que ele aprenda a identificar um código próprio para MCU e, com isso, consiga inferir a sequência ótima de passes. O modelo sofrerá uma leve alteração em parâmetros, porém sua execução e lógica funcional permanecerão as mesmas. Desta forma, o conhecimento geral adquirido em seu treinamento original será herdado e, ao mesmo tempo, atingirá o domínio de otimização para microcontroladores, em específico, aqueles com arquitetura AVR.

O processo empregado para refinamento será o *LLM Supervised Fine-Tuning*, ou SFT (HARADA et al., 2025). Durante o processo de integração de novos hiperparâmetros

[4] Repositório do projeto: <https://github.com/ComputerSystemsLaboratory/YaCoS>

ao modelo (refinamento), iremos ajustar a *learning rate* (taxa de aprendizado), o tamanho do lote passado e outros valores calibrados após algumas séries de experimentação. Cerca de 80% do conjunto de dados preparado será utilizado para o *fine-tunning* do modelo, 10% será empregado para validação e prevenção de *overfitting*, e os 10% restantes serão utilizados para se fazer a inferência e posterior análise comparativa. O treinamento de ajuste fino supervisionado permite que somente as últimas camadas do modelo de inteligência artificial sejam atualizadas, enquanto camadas anteriores são preservadas.

A plataforma de auxílio ao treinamento de inteligência artificial *Hugging Face* será utilizada para execução deste processo, devido à infraestrutura disponibilizada como serviço, próprias para *machine learning*.

3.7 Inferência e Análise Comparativa

Na etapa de análise dos resultados, o modelo será avaliado em dois aspectos principais: sua capacidade de reduzir o tamanho final do código e de manter a compilabilidade dos programas. Serão utilizados códigos do conjunto de validação anteriormente separados do restante destinado ao treinamento, e para cada entrada será inferida uma sequência ótima de passes de otimização pelo modelo. Então, o código será compilado pela sequência de passes fornecida ao final da execução e também pela flag padrão -Oz, nativa de compiladores com base LLVM. Os códigos otimizados gerados serão comparados levando em conta a contagem de instruções e o tamanho de cada um. Ao final, faremos uma média da redução de tamanho percentual obtida pelos passes inferidos pelo modelo em relação à otimização -Oz e à melhor sequência de passes identificada pelo *autotuning*. Todos os experimentos serão realizados de forma automatizada e avaliados de forma quantitativa, de tal modo que o presente estudo avalie com clareza se o modelo trabalhado é ou não capaz de replicar ou superar as otimizações tradicionais em um contexto de arquitetura restrita, AVR.

4 Cronograma

A Tabela 1 apresenta o cronograma de execução da pesquisa considerando períodos quinzenais.

Tabela 1 – Cronograma de Atividades

Atividades	*Período Início*	*Ano*
Geração dos datasets com Csmith e AnghaBench	Julho (2ª quinzena)	2025
Testes do conversor com códigos gerados	Agosto (1ª quinzena)	2025
Geração dos pares prompt/label	Agosto (2ª quinzena)	2025
Organização dos dados para fine-tuning	Setembro (1ª quinzena)	2025
Execução do fine-tuning no Hugging Face	Setembro (2ª quinzena)	2025
Validação dos códigos gerados (comparação com -Oz)	Outubro (1ª quinzena)	2025
Análise quantitativa e documentação dos resultados	Outubro (2ª quinzena)	2025
Escrita dos resultados e revisão do TCC	Novembro (1ª quinzena)	2025
Revisão e defesa do TCC	Novembro (2ª quinzena)	2025

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 11, 14 e 15.

Atmel Corporation. *ATmega640/1280/1281/2560/2561: 8-bit Microcontroller with 256K Bytes In-System Programmable Flash*. [S.l.], 2005. Datasheet, doc2549A–AVR–03/05. Disponível em: <https://ww1.microchip.com/downloads/en/devicedoc/doc2549.pdf>. Citado 2 vezes nas páginas 8 e 17.

BODEN, M. A. Inteligencia artificial. Turner, 2017. Citado na página 20.

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 23.

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 10, 11, 12, 13, 14, 15, 16, 17, 23 e 24.

COSTA, R. H. P. et al. *Compiladores*. [S.l.]: Editora Científica, 2023. ISBN 978-65-00-68043-0. Citado na página 15.

CUMMINS, C. et al. Large language models for compiler optimization. *arXiv preprint arXiv:2309.07062*, 2023. Citado 7 vezes nas páginas 6, 7, 8, 15, 24, 25 e 28.

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 2 vezes nas páginas 25 e 28.

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 10, 12, 13, 14, 16, 17, 23 e 24.

HARADA, Y. et al. Massive supervised fine-tuning experiments reveal how data, layer, and training factors shape llm alignment quality. *arXiv preprint arXiv:2506.14681*, 2025. Citado na página 28.

HUSSAIN, A. et al. Programming a microcontroller. *Int. J. Comput. Appl*, v. 155, n. 5, p. 21–26, 2016. Citado 2 vezes nas páginas 7 e 17.

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 7.

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 24.

LI, Q.; YAO, C. *Real-Time Concepts for Embedded Systems*. Boca Raton, London, New York: CRC Press, Taylor & Francis Group, 2003. Citado na página 7.

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 20 e 21.

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 18.

MORANDÍN-AHUERMA, F. ¿ what is artificial intelligence? 2022. Citado na página 20.

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 20 e 22.

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 8 e 19.

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 25 e 28.

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 19, 20, 21, 22 e 24.

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 27.

The LLVM Project. *LLVM Language Reference Manual*. 2025. <https://llvm.org/docs/LangRef.html>. Acessado em 18 de maio 2025. Citado na página 15.

VASWANI, A. et al. Attention is all you need. *Advances in neural information processing systems*, v. 30, 2017. Citado na página 23.

WANG, Z.; O'BOYLE, M. Machine learning in compiler optimisation. *arXiv preprint arXiv:1805.03441*, 2018. Citado 2 vezes nas páginas 7 e 24.

WHITE, E. *Making Embedded Systems: Design Patterns for Great Software*. [S.l.]: "O'Reilly Media, Inc.", 2024. Citado 3 vezes nas páginas 17, 18 e 19.

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 27.

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 23.