

Review of the Meta Large Language Model Compiler

A Presentation to Vancouver AI

Rudy Cazabon

rudycazabon@outlook.com

February 25, 2025



Agenda

- Presenter Bio and Personal Interest
- What is the Meta Large Language Model Compiler?
- Section 1 – Introduction
- Section 2 – Training Methodology
- Section 3 – Performance Evaluation
- Section 4 – Training Parameters
- Section 5 – Evaluation
- Section 6 – Related Work
- Section 7 – Limitations
- Personal Work, Observations, Directions
- Q&A

Presenter Bio

- Rodolfo "Rudy" Cazabon
- Education
 - BS Space Sciences, Florida Institute of Technology.
 - Graduate Studies in Aerospace Engineering, Georgia Institute of Technology.
- Predominant work in 3D computer graphics (SW/HW) applied to computer games, 3D tools in architectural design, and media & entertainment.
- Veteran of Autodesk, Intel, Adobe, Havok, and Electronic Arts.



Personal Interest

- Interested in exploring whether LLM Compiler (and its relatives) might be a useful in high-performance workflows where managing the combinations of optimization flags and techniques across different hardware architectures can be overwhelming.
- Examples
 - 3D graphics rendering and games.
 - Supercomputing applications, e.g., climate modeling, crash simulations.
 - Realtime responsive systems, e.g., automotive, robotics, aviation.

Meta Large Language Model Compiler: Foundation Models of Compiler Optimization

June 27, 2024

Abstract

<https://doi.org/10.48550/arXiv.2407.02524>

Large Language Models (LLMs) have demonstrated remarkable capabilities across a variety of software engineering and coding tasks. However, their application in the domain of code and compiler optimization remains underexplored. Training LLMs is resource-intensive, requiring substantial GPU hours and extensive data collection, which can be prohibitive. To address this gap, we introduce Meta Large Language Model Compiler (LLM Compiler), a suite of robust, openly available, pre-trained models specifically designed for code optimization tasks. Built on the foundation of Code Llama, LLM Compiler enhances the understanding of compiler intermediate representations (IRs), assembly language, and optimization techniques. The model has been trained on a vast corpus of 546 billion tokens of LLVM-IR and assembly code and has undergone instruction fine-tuning to interpret compiler behavior. LLM Compiler is released under a bespoke commercial license to allow wide reuse and is available in two sizes: 7 billion and 13 billion parameters. We also present fine-tuned versions of the model, demonstrating its enhanced capabilities in optimizing code size and disassembling from x86_64 and ARM assembly back into LLVM-IR. These achieve 77% of the optimising potential of an autotuning search, and 45% disassembly round trip (14% exact match). This release aims to provide a scalable, cost-effective foundation for further research and development in compiler optimization by both academic researchers and industry practitioners.


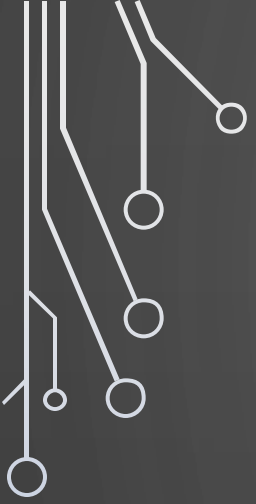
AUTHORS

Written by
[Chris Cummins](#)
Volker Seeker
Dejan Grubisic
[Baptiste Rozière](#)
Jonas Gehring
[Gabriel Synnaeve](#)
Hugh Leather

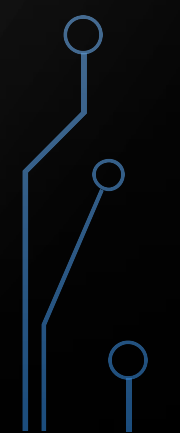

Publisher
ArXiv

What is the Meta Large Language Model Compiler?

- The Meta Large Language Model Compiler (LLM Compiler) are a suite of pre-trained Large Language Models (LLMs) designed for compiler optimization tasks.
- Built on Code Llama aimed to bridge the gap in applying LLMs to code optimization due to the resource-intensive nature of training such models.
- The models are trained on a massive dataset of Low-Level Virtual Machine-Intermediate Representation (LLVM-IR) and ASM code and instruction fine-tuned for compiler behavior.
- The goal is to provide a scalable and cost-effective foundation for future compiler optimization research and development.



Note: The following numbered sections in these slides correspond to sections in the LLM Compiler paper for reference.





Section 1 - Introduction





Introduction

Overview

- Introduction to the Meta Large Language Model (LLM) Compiler.
- Focus on using foundation models for compiler optimization.

Motivation

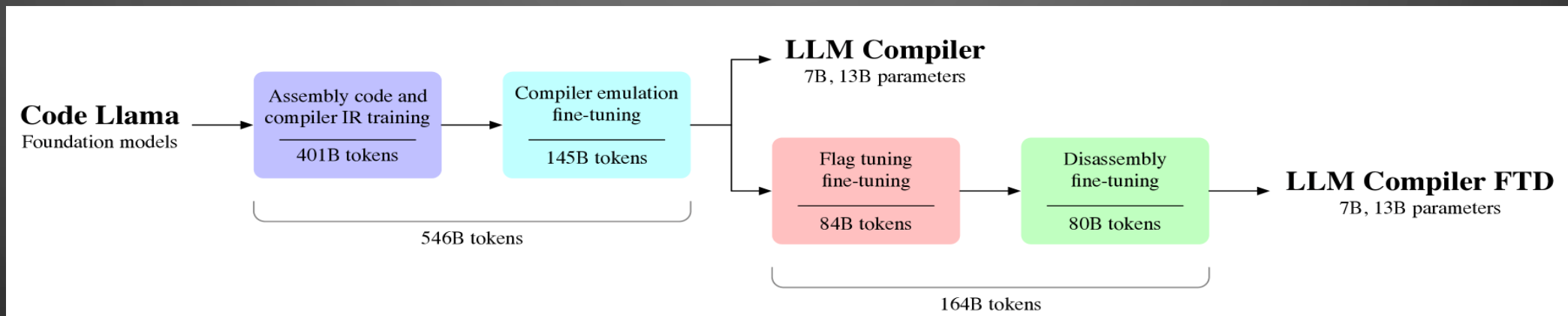
- Traditional compiler optimizations are complex and time-consuming.
- LLMs offer a scalable, automated approach to improving compiler performance.

Contributions

- Introduces an LLM-based compiler optimization framework.
- Benchmarks performance on real-world compiler tasks.
- Provides open-source models for further research.


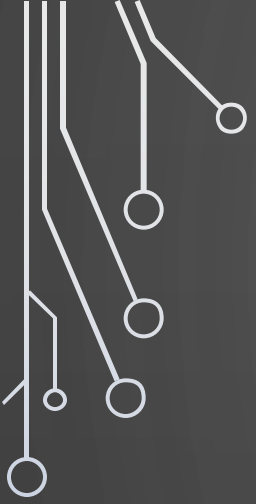
Figure 1 - Overview of LLM Compiler Framework

- Train Code Llama with assembly, compiler IRs, and instruction fine-tuning on a **custom compiler emulation** dataset.
- Goal is to enable the model to better reason about code optimization.

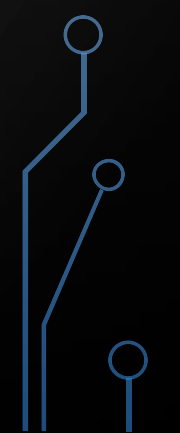



LLM Compiler models are specialized from Code Llama by training on 546 billion tokens of compiler-centric data in two stages.

- **Stage 1 - the models are trained predominantly on unlabeled compiler IRs and assembly code.**
- **Stage 2 - the models are instruction fine-tuned to predict the output and effect of optimizations**



Section 2 – LLM Compiler: Specializing Code Llama for compiler optimization



2.1 Pretraining on assembly code and compiler IRs

- Workflow

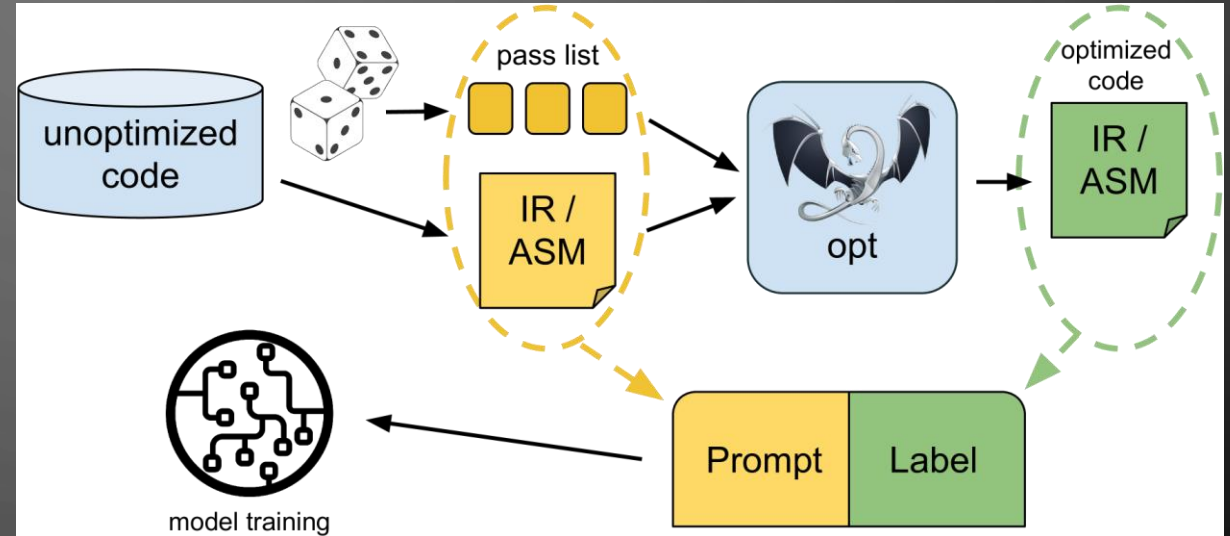
- Start with high-level languages, e.g., Python, C++.
- Build LLM with a good understanding of these languages, initialize LLM Compiler models with the weights of Code Llama.
- Train for 401 billion tokens on a compiler-centric dataset composed mostly of assembly code and compiler IRs.

- Dataset

- LLM compiler trained on compiler IR and ASM generated by LLVM v17.0.6.


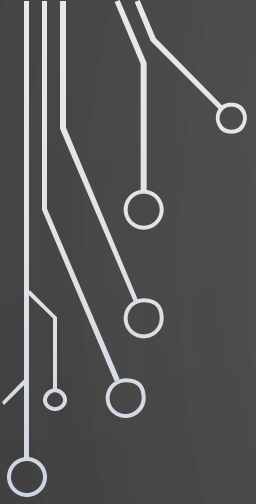
Figure 2 - Training Pipeline for LLM Compiler

- Step 1 - generate from a set of unoptimized seed programs and applying randomly generated sequences of compiler optimizations
- Step 2 - train the model to predict the code generated by the optimizations.
- Step 3 - train the model to predict the code size after the optimizations have been applied.

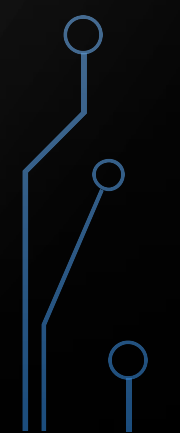



2.2 Instruction fine-tuning for Compiler Emulation

- Workflow
 - Generate as input a set of unoptimized seed programs to which random compiler optimizations are selected.
 - Also train the model to predict the code size after the optimizations have been applied.
- Task specification
 - Given unoptimized LLVM-IR (emitted by Clang++) and ask it to produce a list of **opt** flags that should be applied
 - Measure the binary size & after the optimizations
 - Measure output code.
- Assessment Metrics
 - Measure code size by the number of IR instructions
 - Measure resulting binary size (sum of .TEXT and .DATA sections) of IR or ASM.



Section 3 - LLM Compiler FTD: Extending for downstream compiler tasks



3.1 Instruction fine-tuning for optimization flag tuning

Compiler flags significantly impact runtime performance & code size. LLM Compiler FTD is trained to select optimal flags for LLVM's IR optimization tool opt to minimize code size.

Task Specification

- Model input: Unoptimized LLVM-IR (as emitted by Clang frontend).
- Model output: List of opt flags to apply, before/after binary size, and optimized output code.

Correctness Assurance

- PassListEval tool is developed to detect incorrect pass lists and validate pass lists against self-testing C++ programs and reject pass list if crashes or fails tests.

Dataset & Training Process

Training data:

- Derived from 4.5M unoptimized IRs used for pretraining and pass lists from an iterative compilation process.

Pass List Generation Steps:

- Generated random 50-pass lists per program and evaluated 22 billion compilations.
- Pass List minimization removed redundant passes & commutative orderings and sorted to search for optimal ordering.
- Validation applied PassListEval to identify faulty pass lists.
- Autotuning refinement selected the Top 100 optimal pass lists were shared across programs.

Results & Impact

- Achieved 7.1% binary size reduction over -Oz optimizations.
- Goal: LLM Compiler FTD aims to approach autotuning efficiency without excessive compilations.

Figure 3: Process used to generate training data and how the model is used for inference.

- The model input (Prompt) and output (Label) during training ① and inference ②.
- Generating the label for the training prompt, the unoptimized code is compiled against multiple random pass lists.
- The pass list achieving the minimum binary size is selected, minimized and checked for correctness.
- The final pass list together with its corresponding optimized IR are used as label during training.
- For deployment we generate only the optimization pass list which we feed into the compiler, ensuring that the optimized code is correct.

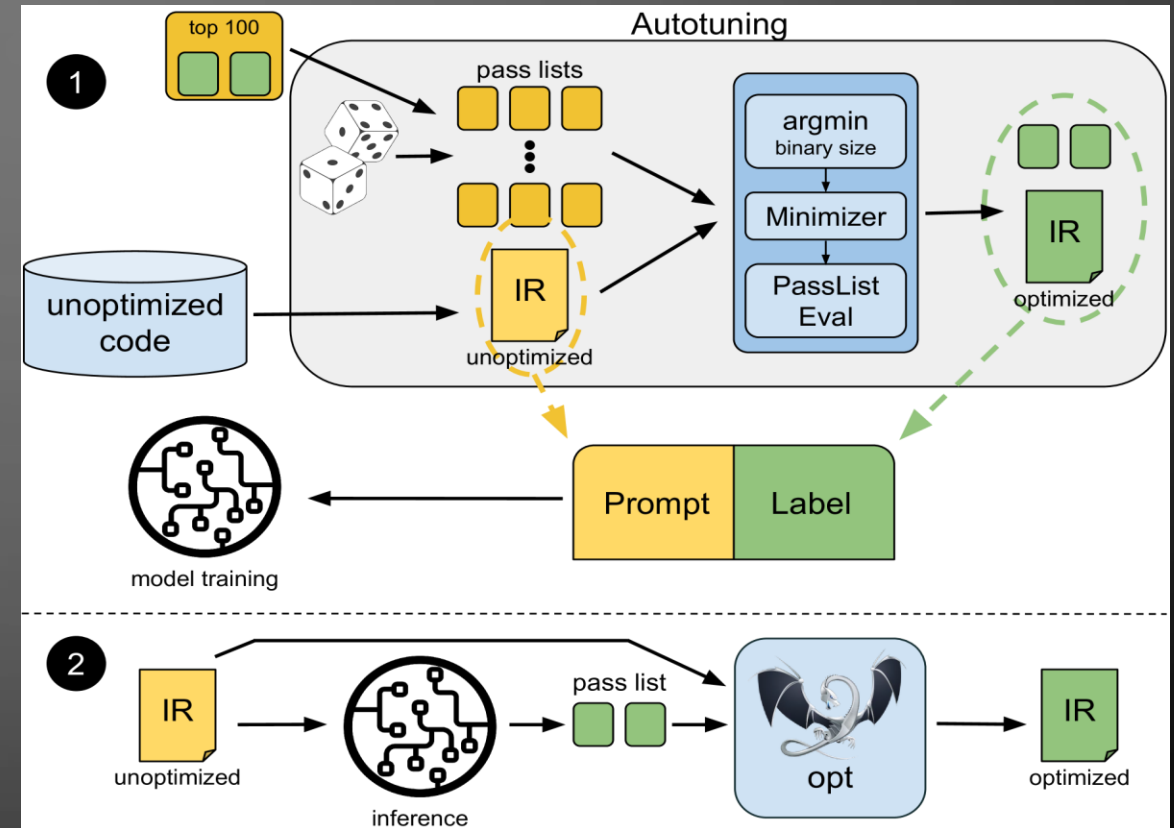
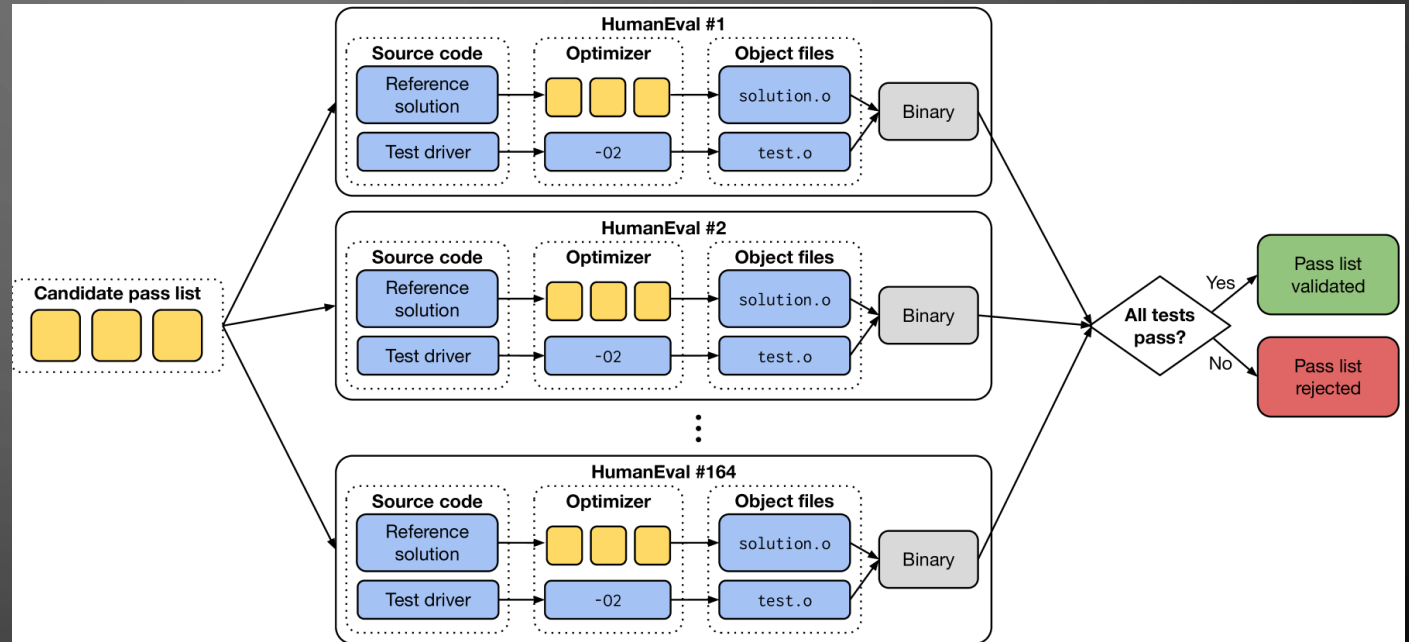


Figure 4: Validating a candidate list of optimization passes using PassListEval.

- The candidate pass list is applied to the reference solutions for all 164 programs in test set.
- The unit tests for these reference solutions are optimized using a conservative -O2 pass pipeline to ensure correctness and then linked against the reference solutions.
- The resulting binaries are executed and if any of the binaries crash during execution, or if any of the compiler invocations fail, the pass list is rejected.



3.2 - Instruction Fine-Tuning for Disassembly

Training Data Generation

- Unoptimized Seed Programs: A diverse set of unoptimized code samples.
- Random Optimization Passes: Generates optimized ASM outputs.
- Paired Datasets Creation: ASM is paired with original LLVM-IR.

Fine-Tuning Process

- Model Input: Takes ASM and predicts the original LLVM-IR.
- Learning Objective: Minimizes the gap between predicted and original LLVM-IR.

Evaluation Metrics

- Compilation Success Rate: Measures syntax correctness.
- Exact Match Rate: Assesses precision in reproducing LLVM-IR.



Section 4 – Training Parameters



Section 4 – Training Parameters

- Tokenizer: Byte Pair Encoding (BPE), same as Code Llama & Llama 2.
- Training Phases: Uses the same training parameters across all four stages.
- Optimizer: AdamW ($\beta_1=0.9$, $\beta_2=0.95$).
- Learning Rate:
 - Cosine schedule with 1000 warm-up steps.
 - Final learning rate = 1/30th of peak learning rate.
 - Initial learning rate: $2e-5$.
- Context Length:
 - Increased from 4,096 to 16,384.
 - Batch size: 4M tokens.
- RoPE (Rotary Positional Embeddings)
 - Modified parameters to support longer context training.
 - Base frequency: $\theta = 10^6$.
 - Matches long-context training from Code Llama.



Section 5 - Evaluation



5.1 Evaluation of LLM Compiler Models Tasks

- Workflow – Evaluate the performance of LLM Compiler models on the tasks of flag tuning, foundation model tasks (compiler emulation and next-token prediction), and software engineering tasks.
- Flag tuning
 - Compare flag tuning for unseen programs and compare to GPT-4 & Code Llama.
 - Run inference for each model, extract optimization pass list and apply to program and record binary size.
- Foundation model tasks
 - Focus on next-token prediction and compiler emulation.
 - Next-token prediction - compute a complexity value on LLVM-IR and ASM code from all optimization levels.
 - Compiler emulation – apply two metrics: do the generated LLVM-IR or ASM compile, and do they exactly match what the compiler would produce.
 - Evaluate at each stage of the training pipeline and record how each successive task affects performance.
- Software engineering tasks
 - Asses whether LLM Compiler FTD (built on Code Llama for software engineering tasks) whether the additional training has affected the performance of code generation.
 - Use same benchmark suite as in Code Llama.



Section 6 – Related Work



Section 6 - Related Work

- Machine learning-guided optimizations
 - Inputs of hand-built code to the ML system with large information loss and failure to reproduce call graph and control flow.
- Language models over code
 - Existing models, e.g., DeepSeek-Coder and GPT-4, are capable of automated program repair and source-level algorithmic optimization but not focused on compilation tasks.
- Language models over IR
 - Identify code weakness but few LLMs include compiler IR in their training.
- Machine Learning in Compilers
 - Mainstay for decades has been compiler pass ordering.



Section 7 – Discussion



Section 7 - Limitations

Capabilities

- Code size optimization.
- Disassembly from x86_64 & ARM ASM to LLVM-IR.

Limitations

- Generalization Issues: Performance may degrade on unseen optimization tasks.
- Disassembly Accuracy: < 50% round-trip rate with only 14% exact matches.
- Compute Requirements: Running and fine-tuning these models requires significant computational resources.
- Lack of Real-World Benchmarking: Further testing is needed to assess performance in diverse compiler settings.
- Limited Support for Non-Traditional Architectures: Models primarily trained on x86_64 and ARM may not perform well on less common architectures.



Personal Work, Observations, Future Directions



Personal Work and Observations

- Executed the “small” model llm-compiler-7b-ftd on the following systems:
 - Google Colab with Nvidia Tesla T4 GPU (16 GB host, 16 GB vRAM).
 - AMD Ryzen9 8495H 32 GB, Nvidia RTX4060 8 GB.
- Difficulties and failures in loading PyTorch model due to overflowing available resources.
- Performed local conversion of PyTorch llm-compiler-7b-ftd model to LlamaCpp GGUF optimized to FP16 with a 42% model size reduction.
- Execution on LlamaCpp was possible but runs were approximately 45 minutes.
- Even when all was working, manually identified compiler missteps such as not reusing registers, no alignment on cache lines, and cache evictions.

Future Personal Directions

- Current LLM Compiler trained on x86 and ARM LLVM-IR and ASM.
- Would the software engineering community be interested:
 - Results for RISC-V architectures?
 - Training and fine-tuning for SPIR-V?
- Interested in conversion to ONNX format and testing with other runtimes such as Triton and DirectML.



Q&A

