Meta Large Language Model Compiler: Foundation Models of Compiler Optimization

Chris Cummins†, Volker Seeker†, Dejan Grubisic, Baptiste Rozière, Jonas Gehring, Gabriel Synnaeve, Hugh Leather†

Meta AI

Abstract

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.

1 Introduction

There is increasing interest in large language models (LLMs) for software engineering tasks including code generation, code translation, and code testing. Models such as StarCoder (Lozhkov et al., 2024), Code Llama (Rozière et al., 2023), and GPT-4 (OpenAI, 2023) have a good statistical understanding of code and can suggest likely completions for unfinished code, making them useful for editing and creating software. However, there is little emphasis on training specifically to optimize code. Publicly available LLMs can be prompted to make minor tweaks to a program such as tagging variables to be stored as registers, and will even attempt more substantial optimizations like vectorization, though they easily become confused and make mistakes, frequently resulting in incorrect code.

Prior works on machine learning-guided code optimization have used a range of representations from hand-built features (Wang & O'Boyle, 2018) to graph neural networks (GNNs) (Liang et al., 2023). However, in all cases, the way the input program is represented to the machine learning algorithm is incomplete, losing some information along the way. For example, Trofin et al. (2021) use numeric features to provide hints for function inlining, but cannot faithfully reproduce the call graph or control flow. Cummins et al. (2021) form graphs of the program to pass to a GNN, but exclude the values of constants and some type information which prevents reproducing instructions with fidelity.

In contrast, LLMs can accept source programs, as is, with a complete, lossless representation. Using text as the input and output representation for a machine learning optimizer has desirable properties: text is a

†: Core contributors.

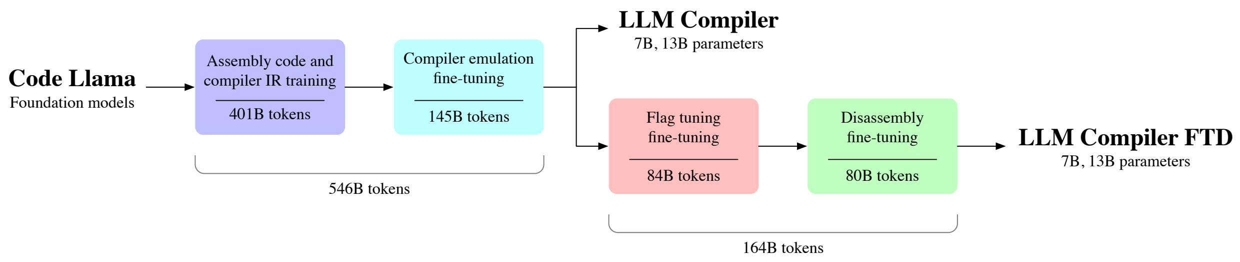


Figure 1: LLM Compiler models are specialized from Code Llama by training on 546 billion tokens of compiler-centric data in two stages. In the first stage the models are trained predominantly on unlabelled compiler IRs and assembly code. In the next stage the models are instruction fine-tuned to predict the output and effect of optimizations. LLM Compiler FTD models are then further fine-tuned on 164 billion tokens of downstream flag tuning and disassembly task datasets, for a total of 710 billion training tokens. During each of the four stages of training, 15% of data from the previous tasks is retained.

universal, portable, and accessible interface, and unlike prior approaches is not specialized to any particular task.

However, training LLMs incurs high cost in both compute and data. For example, training Code Llama's models consumed 1.4M A100 GPU hours to train, and curating the vast amounts of training data (hundreds of billions of tokens) can be challenging. These costs are often prohibitive to researchers in the field and this blocks advances that might otherwise be possible.

To address this issue, we are releasing LLM Compiler, a family of foundation models that have already been trained to understand the semantics of compiler IRs and assemblies and to emulate the compiler, allowing for easy fine-tuning with minimal data for specific downstream compiler optimization tasks. Building upon Code Llama, we extend its capabilities to encompass compiler optimization and reasoning.

The training pipeline for LLM Compiler is illustrated in Figure 1. We extend Code Llama with additional pretraining on a vast corpus of assembly codes and compiler IRs, and then instruction fine-tune on a bespoke *compiler emulation* dataset to better reason about code optimization. Our intention with releasing these models is to provide a foundation for researchers and industry practitioners to further develop code optimization models. We then adapt the models for two downstream compilation tasks: tuning compiler flags to optimize for code size, and disassembling x86_64 and ARM assembly to LLVM-IR. We also release these LLM Compiler FTD models to the community under the same bespoke commercial license. Compared to the autotuning technique on which it was trained, LLM Compiler FTD achieves 77% of the optimizing potential without the need for any additional compilations. When disassembling, LLM Compiler FTD creates correct disassembly 14% of the time. On both tasks LLM Compiler FTD models significantly outperform comparable LLMs Code Llama and GPT-4 Turbo.

Our work aims to establish a scalable, cost-effective foundation for further research and development in compiler optimization, catering to both academic researchers and industry practitioners. By providing access to pre-trained models in two sizes (7 billion and 13 billion parameters) and demonstrating their effectiveness through fine-tuned versions, LLM Compiler paves the way for exploring the untapped potential of LLMs in the realm of code and compiler optimization.

1.1 Overview

Figure 1 shows an overview of our approach. LLM Compiler models target compiler optimization. They are available in two model sizes: 7B and 13B parameters. The LLM Compiler models are initialized with Code Llama model weights of the corresponding size and trained on an additional 546B tokens of data comprising mostly compiler intermediate representations and assembly code. We then further train LLM Compiler FTD models using an additional 164B tokens of data for two downstream compilation tasks: flag tuning and disassembly. At all stages of training a small amount of code and natural language data from previous stages is used to help retain the capabilities of the base Code Llama model.

Dataset	Sampling prop.	Epochs	Disk size
IR and assembly pretraining (401 billion tokens)			
Code	85.00%	1.000	872 GB
Natural language related to code	14.00%	0.019	942 GB
Natural language	1.00%	0.001	938 GB
Compiler emulation (additional 145 billion tokens)			
Compiler emulation	85.00%	1.702	175 GB
Code	13.00%	0.055	872 GB
Natural language related to code	1.80%	0.001	942 GB
Natural language	0.20%	6.9e−5	938 GB
Flag tuning fine-tuning (additional 84 billion tokens)			
Flag tuning	85.00%	1.700	103 GB
Compiler emulation	11.73%	0.136	175 GB
Code	2.84%	0.007	872 GB
Natural language related to code	0.40%	1.1e−4	942 GB
Natural language	0.03%	8.8e−6	938 GB
Disassembly fine-tuning (additional 80 billion tokens)			
Disassembly	85.00%	1.707	88 GB
Flag tuning	4.68%	0.089	103 GB
Compiler emulation	8.07%	0.089	175 GB
Code	1.96%	0.004	872 GB
Natural language related to code	0.27%	7.5e−5	942 GB
Natural language	0.03%	5.7e−6	938 GB

Table 1: Training datasets used.

	Items	Tokens	Disk size
LLVM-IR	10.7 M	185 B	432 GB
Assembly	10.1 M	216 B	440 GB
Total	20.8 M	401 B	872 GB

(a) Language

	Items	Tokens	Disk size
x86_64-unknown-linux-gnu	17.3 M	340.3 B	738 GB
aarch64-unknown-linux-gnu	3.5 M	60.5 B	133 GB
nvptx64-nvidia-cuda	9.2 k	146 M	286 MB
Total	20.8 M	401 B	872 GB

(b) Target

Table 2: Composition of data used for initial IR and assembly pretraining. LLM Compiler is trained on a near-even split of IR and assembly code, predominantly targeting x86-64 architecture, with some 64-bit ARM, and a small amount of CUDA.

2 LLM Compiler: Specializing Code Llama for compiler optimization

2.1 Pretraining on assembly code and compiler IRs

The data used to train coding LLMs are typically composed largely of high level source languages like Python. Assembly code contributes a negligible proportion of these datasets, and compiler IRs even less. To build an LLM with a good understanding of these languages we initialize LLM Compiler models with the weights of Code Llama and then train for 401 billion tokens on a compiler-centric dataset composed mostly of assembly code and compiler IRs, shown in Table 1.

Dataset LLM Compiler is trained predominantly on compiler intermediate representations and assembly code generated by LLVM (Lattner & Adve, 2004) version 17.0.6. These are derived from the same dataset of publicly available code used to train Code Llama. We summarize this dataset in Table 2. As in Code Llama, we also source a small proportion of training batches from natural language datasets.

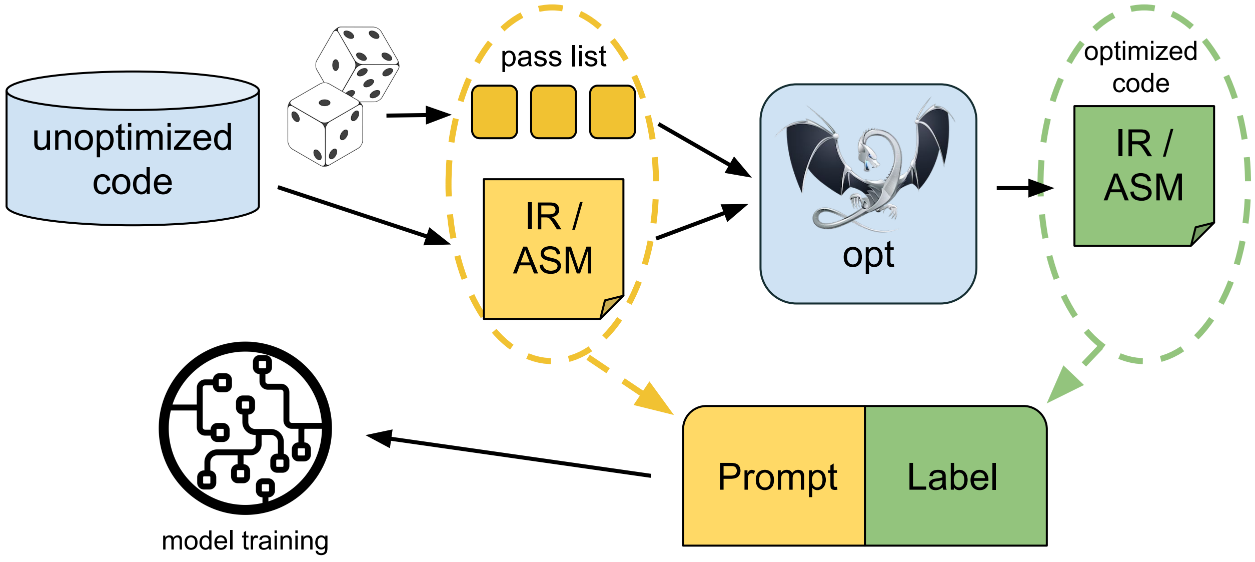


Figure 2: To give the model an understanding of how compiler optimizations work, we use *compiler emulation*. Unoptimized code samples and random pass lists are given to `opt` to generate optimized code (IR or assembly). Pass list and input code are taken together as prompt while the generated output code is used as label.

```
opt input.bc -o output.bc -p 'module(default<Oz>),module(iroutliner)'
clang output.bc -o output.o
size output.o
```

Listing 1: Commands used to apply an optimization pipeline comprising -Oz passes followed by IR outlining to an unoptimized IR `input.bc`. Binary size is the sum of `.TEXT` and `.DATA` section sizes of the lowered object file as reported by `size`.

2.2 Instruction fine-tuning for compiler emulation

To understand the mechanism of code optimization we instruction fine-tune LLM Compiler models to emulate compiler optimizations, illustrated in Figure 2. The idea is to generate from a finite set of unoptimized seed programs a large number of examples by applying randomly generated sequences of compiler optimizations to these programs. We then train the model to predict the code generated by the optimizations. We also train the model to predict the code size after the optimizations have been applied.

Task specification. Given unoptimized LLVM-IR (as emitted by the *clang* frontend), a list of optimization passes, and a starting code size, generate the resulting code after those optimizations have been applied and the resulting code size.

There are two flavors of this task: in the first the model is expected to output compiler IR, in the second the model is expected to output assembly code. The input IR, optimization passes, and code size are the same for both flavors. The prompt dictates the required output format. Examples of each prompt are provided in Appendices Listings 2 and 3.

Code size. We use two metrics for code size: the number of IR instructions, and *binary size*. Binary size is computed by summing the size of the `.TEXT` and `.DATA` sections of the IR or assembly after lowering to an object file; we exclude `.BSS` section from our binary size metric since it does not affect on-disk size.

Optimization passes. In this work we target LLVM 17.0.6 and use the New Pass Manager (PM, 2021) which classifies passes for different levels such as *module*, *function*, *loop*, etc. as well as transformation and analysis passes. Transformation passes change given input IR while analysis passes generate information that influence subsequent transformations.

Of the 346 possible pass arguments for *opt*, we select 167 to use. This includes each of the default optimization pipelines (e.g. `module(default<Oz>)`), individual optimization transform passes (e.g. `module(constmerge)`), but excludes non-optimization utility passes (e.g. `module(dot-callgraph)`) and transformations passes that are not semantics preserving (e.g. `module(internalize)`). We exclude analysis passes since they have no side effects and we rely on the pass manager to inject dependent analysis passes as needed. For passes that accept parameter arguments we use the default values (e.g. `module(licm<allowspeculation>)`). Table 9 contains a list of all passes used. We used LLVM's *opt* tool to apply pass lists and *clang* to lower the resulting IR to object file. Listing 1 shows the commands used.

Dataset. We generated the compiler emulation dataset by applying random lists of between 1 and 50 optimization passes to unoptimized programs summarized in Table 2. The length of each pass list was selected uniformly at random. Pass lists were generated by uniformly sampling from the set of 167 passes described above. Pass lists which resulted in compiler crashes or timed out after 120 seconds were excluded.

3 LLM Compiler FTD: Extending for downstream compiler tasks

3.1 Instruction fine-tuning for optimization flag tuning

Manipulating compiler flags is well known to have a considerable impact on both runtime performance and code size (Fursin et al., 2005). We train LLM Compiler FTD models on the downstream task of selecting flags for LLVM's IR optimization tool *opt* to produce the smallest code size. Machine learning approaches to flag tuning have shown good results previously, but struggle with generalizing across different programs (Cummins et al., 2022). Previous works usually need to compile new programs tens or hundreds of times to try out different configurations and find out the best-performing option. We train and evaluate LLM Compiler FTD models on the zero-shot version of this task by predicting flags to minimize code size of unseen programs. Our approach is agnostic to the chosen compiler and optimization metric, and we intend to target runtime performance in the future. For now, optimizing for code size simplifies the collection of training data.

Task specification. We present the LLM Compiler FTD models with an unoptimized LLVM-IR (as emitted by the *clang* frontend) and ask it to produce a list of *opt* flags that should be applied to it, the binary size before and after these optimizations are applied, and the output code. If no improvement can be made over the input code, a short output message is generated that contains only the unoptimized binary size. Listings 4 and 5 provide the prompt and output templates for this task.

We used the same constrained set of optimization passes as in the compiler emulation task, and compute binary size in the same manner.

Figure 3 illustrates the process used to generate training data (described below) and how the model is used for inference. Only the generated pass list is needed at evaluation time. We extract the pass list from the model output and run *opt* using the given arguments. We can then evaluate the accuracy of the model predicted binary sizes and optimized output code, but those are auxiliary learning tasks not required for use.

Correctness. LLVM's optimizer is not free from bugs and running optimization passes in unexpected or untested orders may expose subtle correctness errors that undermine the utility of the model. To mitigate this risk we developed *PassListEval*, a tool to help in automatically identifying pass lists that break program semantics or cause compiler crashes. An overview of the tool is shown in Figure 4. PassListEval accepts as input a candidate pass list and evaluates it over a suite of 164 self-testing C++ programs, taken from HumanEval-X (Zheng et al., 2023). Each program contains a reference solution for a programming challenge, e.g. *"Check if in given vector of numbers, are any two numbers closer to each other than given threshold"*, and a suite of unit tests that validate correctness. We apply the candidate pass lists to the reference solution, and then link them against the test suites to produce a binary. When executed, the binary will crash if any of the tests fail. If any binary crashes, or if any of the compiler invocations fail, we reject the candidate pass list.

Dataset. We trained LLM Compiler FTD models on a dataset of flag tuning examples derived from 4.5M of the unoptimized IRs used for pretraining. To generate the example optimal pass list for each program we ran an extensive iterative compilation process depicted in Figure 3 and outlined below:

1. We used large-scale *random search* to generate an initial candidate best pass list for the programs. For each program we independently generated random lists of up to 50 passes by uniformly sampling from the set of 167 searchable passes described previously. Every time we evaluated a pass list on a program we recorded

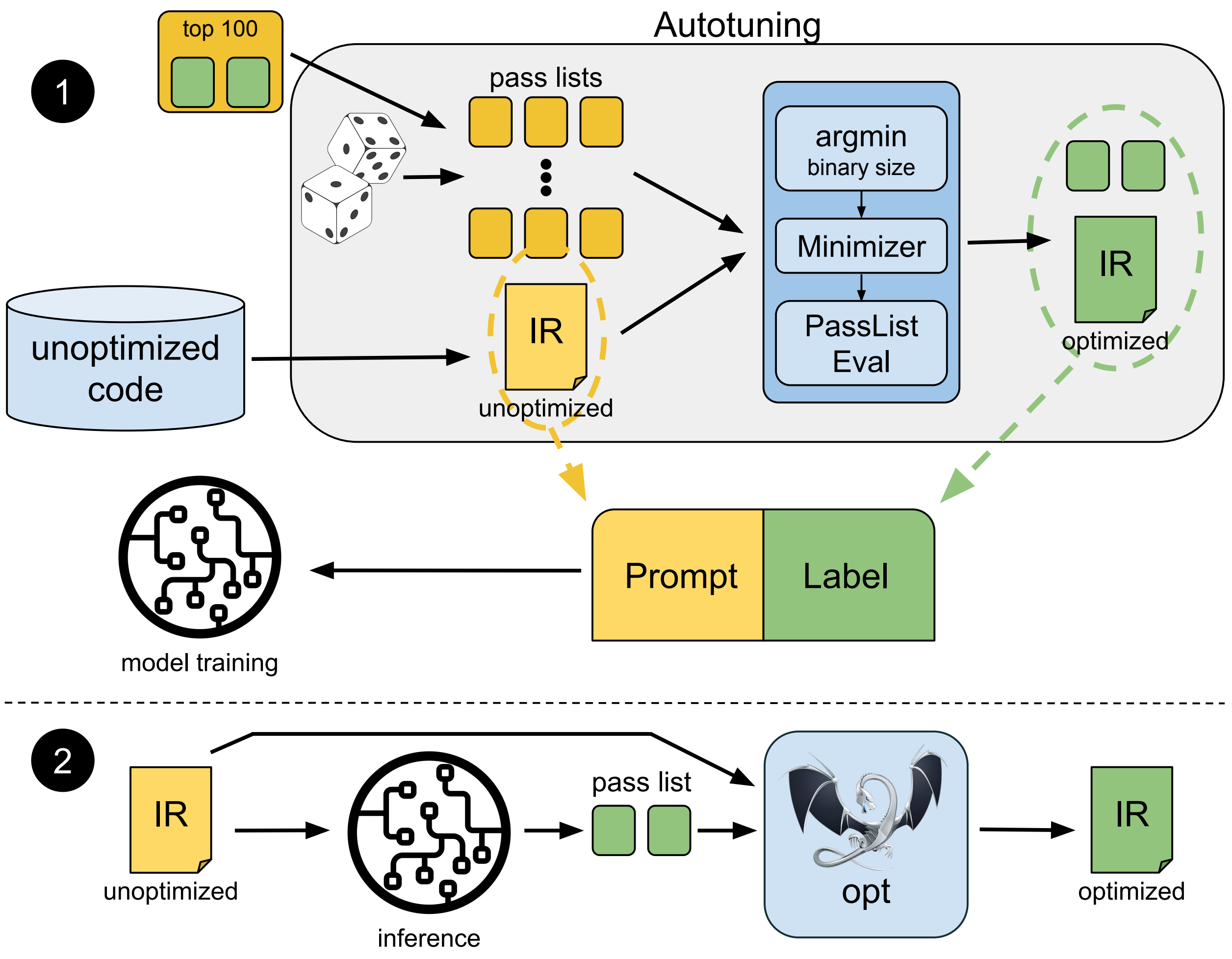


Figure 3: Overview of our approach, showing the model input (Prompt) and output (Label) during training ① and inference ②. The prompt contains unoptimized code. The label contains an optimization pass list, binary size, and the optimized code. To generate the label for the training prompt, the unoptimized code is compiled against multiple random pass lists. The pass list achieving the minimum binary size is selected, minimized and checked for correctness with PassListEval. The final pass list together with its corresponding optimized IR are used as label during training. In a last step, the top 100 most often selected pass lists are broadcast among all programs. For deployment we generate only the optimization pass list which we feed into the compiler, ensuring that the optimized code is correct.

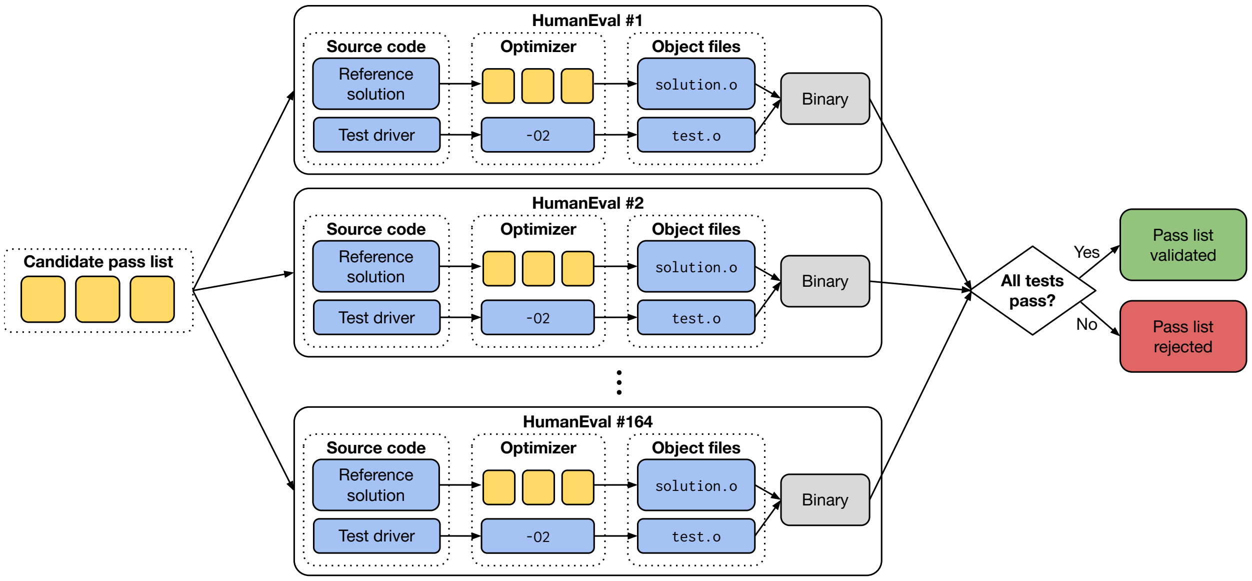


Figure 4: Validating a candidate list of optimization passes using PassListEval. The candidate pass list is applied to the reference solutions for all 164 programs in HumanEval-X. The unit tests for these reference solutions are optimized using a conservative -O2 pass pipeline to ensure correctness, 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.

the resulting binary size. We then pick the per-program pass lists that produced the lowest binary size. We ran 22 billion unique compilations for an average 4,877 per program.

2. The pass lists generated by random search may contain redundant passes that have no effect on the final outcome. Further, some pass orderings are commutative such that reordering then does not affect the final outcome. Since these would introduce noise in our training data, we developed a *minimization* process which we applied to each pass list. Minimization comprises three steps: redundant pass elimination, bubble sort, and insertion search. In redundant pass elimination we minimize the best pass list by iteratively removing individual passes to see if they contribute to the binary size. If not, they are discarded. This is repeated until no further passes can be discarded. Bubble sort then attempts to provide a uniform ordering for pass subsequences by sorting passes based on a key. Finally, insertion sort performs a local search by iterating over each pass in the pass list and attempting to insert each of the 167 search passes before it. If doing so improves the binary size, this new pass list is kept. The entire minimization pipeline loops until a fixed point is reached. The distribution of minimized pass list lengths is shown in Figure 9. The average pass list length is 3.84.
3. We apply *PassListEval*, described previously, to the candidate best pass lists. Through this we identified 167,971 of 1,704,443 unique pass lists (9.85%) as causing compile time or runtime errors.
4. We broadcast the *top 100* most frequently optimal pass lists across all programs, updating the per-program best pass lists if improvements are found. After this the total number of unique best pass lists decreases from 1,536,472 to 581,076.

The autotuning pipeline outlined above produced a geometric mean 7.1% reduction in binary size over -Oz. Figure 10 shows the frequency of individual passes. For our purposes, this autotuning serves as a gold standard for the optimization of each program. While the binary size savings discovered are significant, this required 28 billion additional compilations at a computational cost of over 21,000 CPU days. The goal of instruction fine-tuning LLM Compiler FTD to perform the flag tuning task is to achieve some fraction of the performance of the autotuner without requiring running the compiler thousands of times.

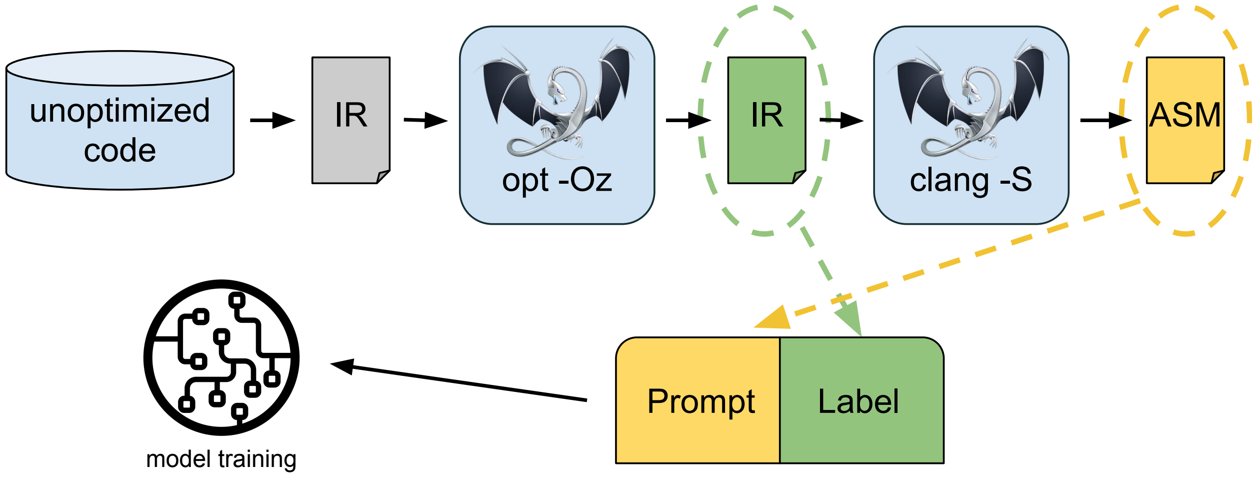


Figure 5: We train the model to understand the relationship between assembly and IR by training it to disassemble a given code sample to its corresponding IR. The IR used to label this training task was generated by optimizing an IR with the -Oz flag.

3.2 Instruction fine-tuning for disassembly

The ability to lift code from assembly back into higher level structures enables running additional optimizations on library code directly integrated with application code or porting of legacy code to new architectures. The field of decompilation has seen advancements in applying machine learning techniques to generate readable and accurate code from binary executables. Several studies explore the use of machine learning for decompilation tasks, such as lifting binaries into intermediate representations for evaluation against synthetic C programs (Cao et al., 2022), utilizing evolutionary approaches like genetic algorithms for program analysis (Schulte et al., 2018), and proposing methods like XLIR for matching binary code across different programming languages (Gui et al., 2022). Armengol-Estapé et al. (2024) have trained a language model to decompile x86 assembly into high level C code. In this study, we demonstrate how LLM Compiler FTD can learn the relationship between assembly code and compiler IR by fine-tuning it for disassembly. The task is to learn the inverse translation of `clang -xir - -o - -S`, shown in Figure 5.

Round tripping. Using an LLM for disassembly causes problems of correctness. The lifted code must be verified by an equivalence checker which is not always feasible or manually verified for correctness or subjected to sufficient test cases to give confidence. However, a lower bound on correctness can be found by round-tripping. That is to say by compiling the lifted IR back into assembly, if the assembly is identical then the IR is correct. This gives an easy route to using the results of the LLM and an easy way to measure the utility of a disassembly model.

Task specification. We provide the model with assembly code and train it to emit the corresponding disassembled IR. Listing 7 shows the prompt format. The context length for this task is set to 8k tokens for the input assembly code and 8k tokens for the output IR.

Dataset. We derive the assembly codes and IR pairs from the same dataset used in previous tasks. Our fine-tuning dataset consists in 4.7M samples. The input IR has been optimized with -Oz before being lowered to x86 assembly.

4 Training parameters

Data is tokenized via byte pair encoding (Gage, 1994), employing the same tokenizer as Code Llama, Llama (Touvron et al., 2023a), and Llama 2 (Touvron et al., 2023b).

We use the same training parameters for all four stages of training. Most of the training parameters we used are the same as for the Code Llama base model. We use the AdamW (Loshchilov & Hutter, 2017) optimizer with β_1 and β_2 values of 0.9 and 0.95. We use a cosine schedule with 1000 warm-up steps, and set the final learning rate to be 1/30th of the peak learning rate. Compared to the Code Llama base model, we increased the context length of individual sequences from 4,096 to 16,384, but kept the batch size constant at 4M tokens. To account for the longer context, we set our learning rate to $2e^{-5}$ and modified the parameters of the RoPE positional embeddings (Su et al., 2024) where we reset frequencies with a base value of $\theta = 10^6$. These settings are in accordance with the long context training done for the Code Llama base model.

5 Evaluation

In this section we evaluate the performance of LLM Compiler models on the tasks of flag tuning and disassembly, compiler emulation, next-token prediction, and finally software engineering tasks.

5.1 Flag tuning task

Methodology. We evaluate LLM Compiler FTD on the task of optimization flag tuning for unseen programs and compare to GPT-4 Turbo and Code Llama - Instruct. We run inference on each model and extract from the model output the optimization pass list. We then use this pass list to optimize the particular program and record the binary size. The baseline is the binary size of the program when optimized using -Oz.

For GPT-4 Turbo and Code Llama - Instruct we append a suffix to the prompt with additional context to further describe the problem and expected output format. After some experimentation we found that the prompt suffix shown in Listing 6 provides the best performance.

All model-generated pass lists are validated using *PassListEval*, and -Oz is used as substitute if validation fails. To further validate correctness of model-generated pass lists we link the final program binaries and differential test their outputs against the outputs of the benchmark when optimized using a conservative -O2 optimization pipeline.

Dataset. We evaluate on 2,398 test prompts extracted from the MiBench benchmark suite (Guthaus et al., 2001). To generate these prompts we take all of the 713 translation units that make up the 24 MiBench benchmarks and generate unoptimized IRs from each. We then format them as prompts as per Listing 4. If the resulting prompt exceeds 15k tokens we split the LLVM module representing that translation unit into smaller modules, one for each function, using *llvm-extract*. This results in 1,985 prompts which fit within the 15k token context window, leaving 443 translation units which do not fit. We use -Oz when for the 443 excluded translation units when computing performance scores. Table 10 summarizes the benchmarks.

Results. Table 3 shows zero-shot performance of all models on the flag tuning task. Only LLM Compiler FTD models provide an improvement over -Oz, with the 13B parameter model marginally outperforming the smaller model, generating smaller object files than -Oz in 61% of cases.

In some cases the model-generated pass list causes a larger object file size than -Oz. For example, LLM Compiler FTD 13B regresses in 12% of cases. These regressions can be avoided by simply compiling the program twice: once using the model-generated pass list, once using -Oz, and selecting the pass list which produces the best result. By eliminating regressions wrt -Oz, these *-Oz backup* scores raise the overall improvement over -Oz to 5.26% for LLM Compiler FTD 13B, and enable modest improvements over -Oz for Code Llama - Instruct and GPT-4 Turbo. Figure 6 shows the performance of each model broken down by individual benchmark.

Binary size accuracy. While the model-generated binary size predictions have no effect on actual compilation, we can evaluate the performance of the models at predicting binary sizes before and after optimization to give an indication of each model's understanding of optimization. Figure 7 shows the results. LLM Compiler FTD binary size predictions correlate well with ground truth, with the 7B parameter model achieving MAPE values of 0.083 and 0.225 for unoptimized and optimized binary sizes respectively. The 13B parameter model improved has similar MAPE values of 0.082 and 0.225. Code Llama - Instruct and GPT-4 Turbo binary size predictions show little correlation with ground truth. We note that the LLM Compiler FTD errors are slightly higher for optimized code than unoptimized code. In particular, there is an occasional tendency for LLM Compiler FTD to overestimate the effectiveness of optimization, resulting in a lower predicted binary size than actual.

Ablation studies. Table 4 ablates the performance of models on a small holdout validation set of 500 prompts taken from the same distribution as our training data (though not used during training). We trained for flag tuning at each stage of the training pipeline from Figure 1 to compare performance. As shown, disassembly training causes a slight regression in performance from average 5.15% to 5.12% improvement over -Oz. We also show performance of the autotuner used for generating the training data described in Section 2. LLM Compiler FTD achieves 77% of the performance of the autotuner.

Table 3: Comparison of model performance when flag tuning 2,398 object files from MiBench. *Overall improvement* scores include 443 object files which do not fit in the context window of LLM Compiler FTD. For GPT-4 and the Code Llama models we appended a suffix to the prompt to provide additional context (see Listing 6 in the Appendix).

	Size	Improved	Regressed	Overall improvement over -Oz zero-shot	Overall improvement over -Oz -Oz backup
LLM Compiler FTD	7B	1,465	302	4.77%	5.24%
	13B	**1,466**	**299**	**4.88%**	**5.26%**
Code Llama - Instruct	7B	379	892	-0.49%	0.23%
	13B	319	764	-0.42%	0.18%
	34B	230	493	-0.27%	0.15%
GPT-4 Turbo (2024-04-09)	-	13	24	-0.01%	0.03%

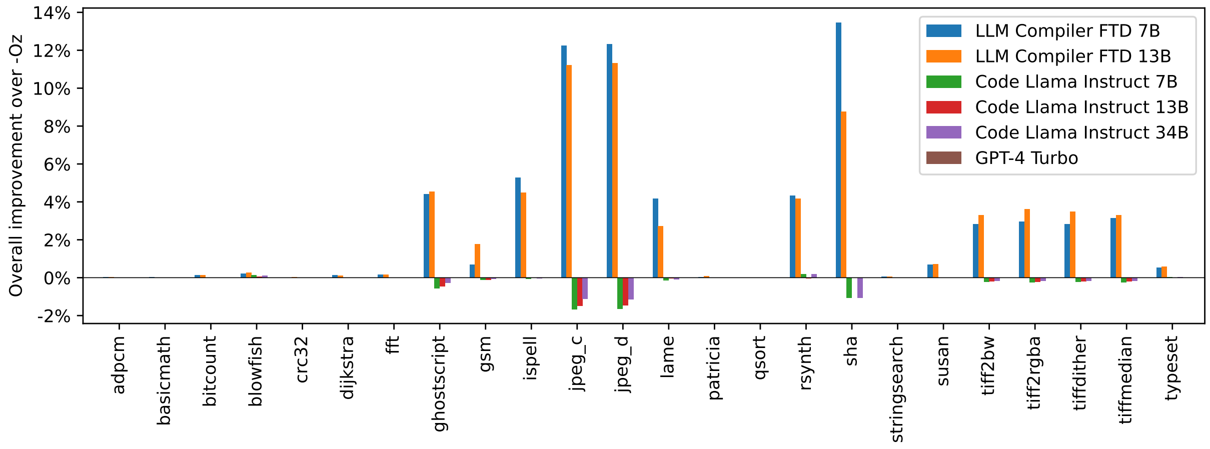
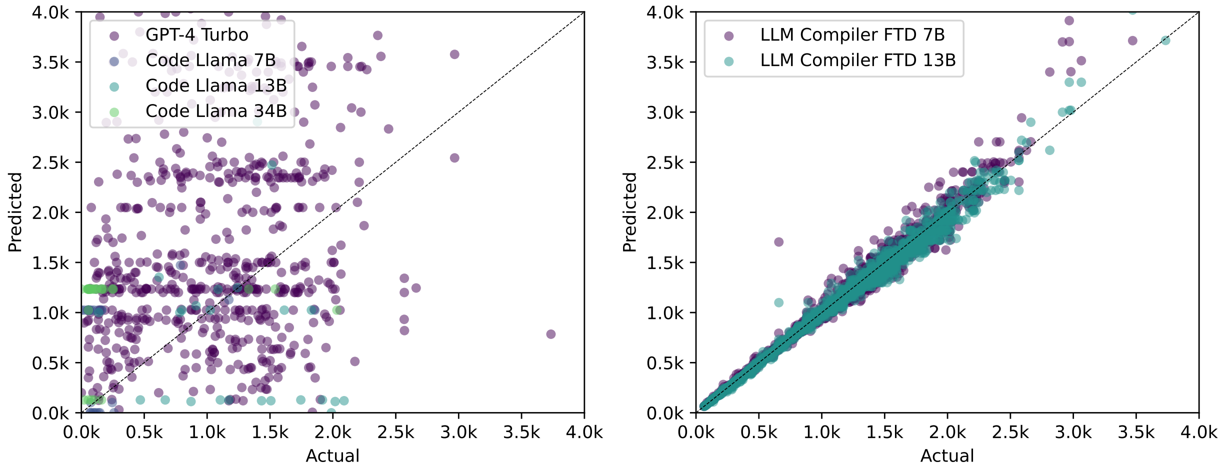


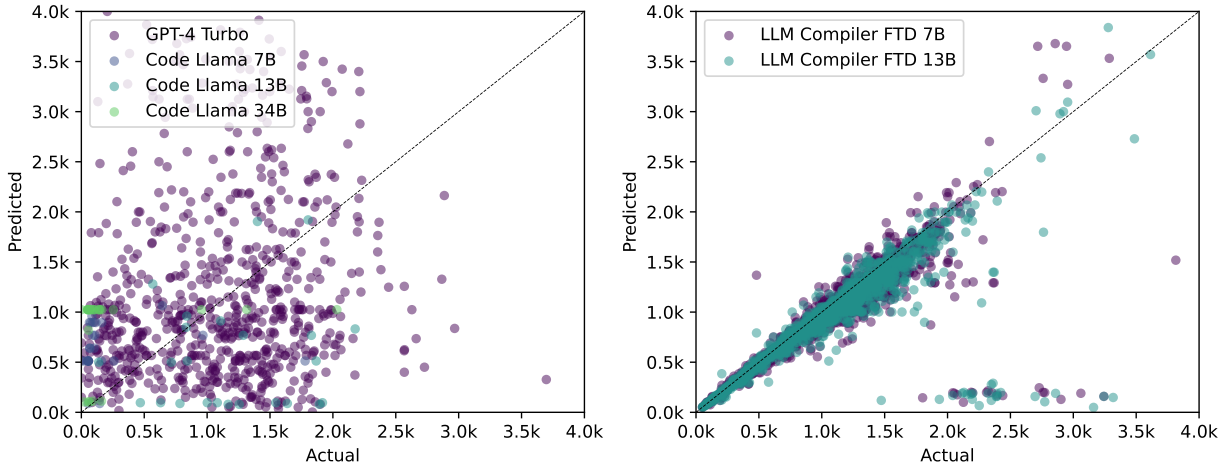
Figure 6: Improvement over -Oz for each of the benchmarks in MiBench.

Table 4: Ablating the LLM Compiler FTD training regime on the flag tuning task. All results are for 7B parameter models, evaluated on the same holdout validation set of 500 programs. The first row is the LLM Compiler FTD release. All other rows strip out successful components of the training regime.

Code Llama	IR & asm pretraining	Compiler emulation	Flag tuning	Disassembly	Mean improvement over -Oz	Mean improvement wrt. Autotuner
✓	✓	✓	✓	✓	5.12%	77%
✓	✓	✓	✓		**5.15%**	**78%**
✓	✓		✓		5.07%	76%
✓			✓		4.94%	75%
			✓		4.79%	72%
Autotuner					6.63%	100%



(a) Unoptimized binary size



(b) Optimized binary size

Figure 7: Accuracy of models at predicting code size before and after optimization. LLM Compiler FTD is most accurate at predicting code size before optimization than after optimization. Code Llama and GPT-4 Turbo, shown left, display little correlation between predicted and actual values.

Table 5: Model performance at disassembling 2,015 assembly codes taken from MiBench. We use *Round trips* to evaluate the capabilities of models, by taking the IR generated by the models and attempting to lower it back to assembly. *Round trips* shows the number of disassembled IRs that can be lowered back, *Round trip BLEU* compares the round-tripped assemblies against the originals, and *Round trip exact match* is the proportion of round-tripped assemblies that are exact character-for-character matches with the input, indicating lossless round-trip from assembly up to IR and back down again.

	Size	Round trips	Round trip BLEU	Round trip exact match
LLM Compiler FTD	7B	**936**	0.951	12.7%
	13B	905	**0.960**	**13.8%**
Code Llama - Instruct	7B	30	0.477	0.0%
	13B	53	0.615	0.0%
	34B	12	0.458	0.0%
GPT-4 Turbo (2024-04-09)	-	127	0.429	0.0%

Table 6: Ablating the LLM Compiler FTD training regime on code disassembly. All results are for 7B parameter model sizes, evaluated on a holdout validation set of 500 programs. Values in parentheses show relative performance to the first row (i.e. the LLM Compiler FTD release).

Code Llama	IR & asm pretraining	Compiler emulation	Flag tuning training	Disassembly	Round trips	Round trip BLEU
✓	✓	✓	✓	✓	**49.4% (-)**	0.951 (-)
✓	✓	✓		✓	45.2% (-8.5%)	0.955 (+0.4%)
✓	✓			✓	44.2% (-10.5%)	0.957 (+0.7%)
✓				✓	39.0% (-21.1%)	**0.965 (+1.5%)**
				✓	8.8% (-82.8%)	0.908 (-4.5%)

5.2 Disassembly task

Methodology. We evaluate the functional correctness of LLM-generated code when disassembling assembly code to LLVM-IR. As in Section 5.1 we evaluate LLM Compiler FTD and compare to Code Llama - Instruct and GPT-4 Turbo, and find that an additional prompt suffix, shown in Listing 8, is required to extract the best performance from these models. The suffix provides additional context about the task and the expected output format. To evaluate the performance of models we *round-trip* the model-generated disassembled IR back down to assembly. This enables us to evaluate accuracy of the disassembly by comparing the BLEU score (Papineni et al., 2002) of the original assembly against the round-trip result. A lossless and perfect disassembly from assembly to IR will have a round-trip BLEU score of 1.0 (*exact match*).

Dataset. We evaluate on 2,015 test prompts extracted from the MiBench benchmark suite. We took the 2,398 translation units used for the flag tuning evaluation above and generated disassembly prompts. We then filtered the prompts on a maximum 8k token length, allowing 8k tokens for the model output, leaving 2,015. Table 11 summarizes the benchmarks.

Results. Table 5 shows performance of the models on the disassembly task. LLM Compiler FTD 7B has a slightly higher round-trip success rate than LLM Compiler FTD 13B, but LLM Compiler FTD 13B has the highest accuracy of round-tripped assembly (*round trip BLEU*) and most frequently produces a perfect disassembly (*round trip exact match*). Code Llama - Instruct and GPT-4 Turbo struggle with generating syntactically correct LLVM-IR. Figure 8 shows the distribution of round-trip BLEU scores for all models.

Ablation studies. Table 6 ablates the performance of models on a small holdout validation set of 500 prompts taken from the MiBench dataset used previously. We trained for disassembly at each stage of the training pipeline from Figure 1 to compare performance. Round trip rate is highest when going through the whole stack of training data and drops consistently with every training stage, though round trip BLEU varies little with each stage.

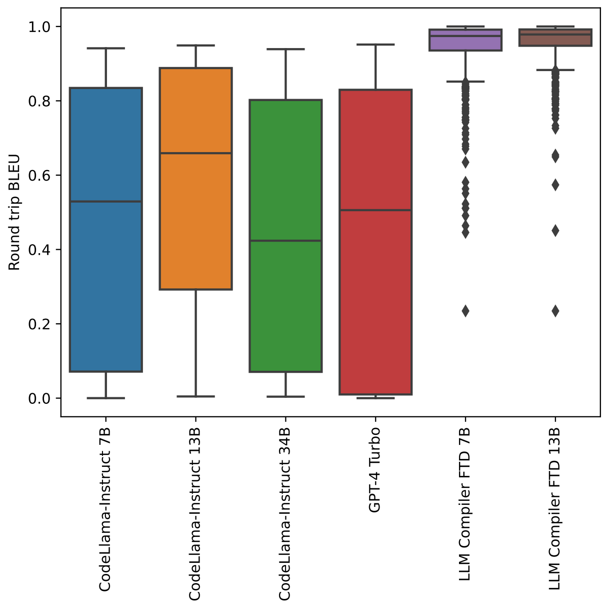


Figure 8: Distribution of round trip BLEU scores on the disassembly task.

5.3 Foundation model tasks

Methodology We ablate LLM Compiler models on the two foundation model tasks of next-token prediction and compiler emulation. We perform this evaluation at each stage of the training pipeline to see how training for each successive task affects performance. For next-token prediction we compute perplexity on a small sample of LLVM-IR and assembly code from all optimization levels. We evaluate compiler emulation using two metrics: whether the generated IR or assembly code compiles, and whether the generated IR or assembly code is an exact match for what the compiler would produce.

Dataset. For next-token prediction we use a small holdout set of validation data that is drawn from the same distribution as our training data but has not been used for training. We use a mixture of optimization levels including unoptimized code, code optimized with -Oz, and randomly generated pass lists. For compiler emulatino we evaluate using 500 prompts generated from MiBench using randomly pass lists generated in the manner described in Section 2.2.

Results Table 7 shows performance of LLM Compiler FTD across all training stages on the two foundation model training tasks of next-token prediction and compiler emulation. Next-token prediction performance jumps sharply after Code Llama, which has seen very little IR and assembly, and declines slightly with each subsequent stage of fine-tuning.

For compiler emulation, the Code Llama base model and the pre-trained models perform poorly since they have not been trained on this task. The highest performance is achieved directly after compiler emulation training where 95.6% of IR and assembly generated by LLM Compiler FTD 13B compiles, and 20% of it matches the compiler exactly. Performance declines after fine-tuning for flag tuning and disassembly.

5.4 Software engineering tasks

Methodology. While the purpose of LLM Compiler FTD is to provide foundation models for code optimization, it builds upon base Code Llama models which were trained for software engineering tasks. To evaluate how the additional training of LLM Compiler FTD has affected the performance of code generation we use the same benchmark suites as in Code Llama that evaluate the ability of LLMs to generate Python code from natural language prompts, such as *“Write a function to find the longest chain which can be formed from the given set of pairs.”*.

Table 7: Performance at next-token prediction and compiler emulation tasks. For *Perplexity*, lower is better. For *Compiles* and *Exact match*, higher is better.

Code Llama	IR & asm pretraining	Compiler emulation	Flag tuning	Disassembly	Size	Perplexity		Compiler emulation	
						IR	Asm	Compiles	Exact match
✓					7B	1.456	1.423	5.4%	1.2%
					13B	1.429	1.404	4.8%	0.8%
✓	✓				7B	1.050	1.041	0.8%	0.0%
					13B	**1.045**	**1.038**	35.8%	2.8%
✓	✓	✓			7B	1.052	1.046	87.0%	16.0%
					13B	1.047	1.043	**95.6%**	**20.0%**
✓	✓	✓	✓		7B	1.058	1.051	55.0%	1.2%
					13B	1.052	1.048	58.6%	4.2%
✓	✓	✓	✓	✓	7B	1.057	1.053	71.0%	4.6%
					13B	1.054	1.052	61.4%	5.4%

Table 8: Performance on Python programming tasks. pass@1 are computed with greedy decoding. The pass@10 and pass@100 scores are computed with nucleus sampling with p=0.95 and temperature=0.6.

Code Llama	IR & asm pretraining	Compiler emulation	Flag tuning	Disassembly	Size	HumanEval			MBPP		
						pass@1	pass@10	pass@100	pass@1	pass@10	pass@100
✓					7B	32.9%	63.3%	85.3%	45.4%	67.5%	81.6%
					13B	**36.0%**	**71.9%**	**90.6%**	**48.4%**	**71.3%**	**83.9%**
✓	✓				7B	28.0%	58.6%	84.3%	42.8%	66.0%	80.0%
					13B	34.1%	68.0%	87.9%	47.6%	70.3%	83.3%
✓	✓	✓			7B	25.0%	51.3%	79.0%	37.4%	61.5%	75.6%
					13B	31.1%	62.9%	83.2%	46.0%	67.8%	80.9%
✓	✓	✓	✓		7B	24.4%	46.2%	73.1%	36.6%	58.5%	74.4%
					13B	29.3%	55.9%	81.1%	42.2%	63.6%	79.1%
✓	✓	✓	✓	✓	7B	26.8%	44.0%	65.3%	31.4%	55.1%	73.2%
					13B	25.6%	51.2%	76.8%	37.6%	60.6%	76.4%
Llama 2					7B	12.2%	25.2%	44.4%	20.8%	41.8%	65.5%
					13B	20.1%	34.8%	61.2%	27.6%	48.1%	69.5%

Datasets. We use the HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021) benchmarks as in Code Llama.

Results. Table 8 shows the greedy decoding performance (*pass@1*) of all model training stages and model sizes starting at the Code Llama base model. It also shows the models' scores on *pass@10* and *pass@100* which were generated with p=0.95 and temperature=0.6. Each stage of compiler-centric training causes a slight regression in Python programming ability. pass@1 performance on HumanEval and MBPP declines by up to 18% and 5% for LLM Compiler and by up to 29% and 22% for LLM Compiler FTD after the additional flag tuning and disassembly fine-tuning. All models still outperform Llama 2 on both tasks.

6 Related work

Language models over code. There is increasing interest in LLMs for source code reasoning and generation (Jiang et al., 2024; Hou et al., 2023). The main enablers of progress in this area are pretrained foundational models made available for others to build upon, including Code Llama (Rozière et al., 2023), StarCoder (Lozhkov et al., 2024), Magicoder (Wei et al., 2024), DeepSeek-Coder (Guo et al., 2024), GPT-4 (OpenAI, 2023) and others (Wang et al., 2023; Allal et al., 2023; Feng et al., 2020). Some of the existing models are open source (Rozière et al., 2023; Lozhkov et al., 2024; Wei et al., 2024; Allal et al., 2023) while others are closed source (Chen et al., 2021; OpenAI, 2023; Li et al., 2022; Gunasekar et al., 2023). We extend the collection of foundational models for code with a family of models specifically trained on intermediate code representation with a license that allows wide reuse.

Language models have been adapted to perform program fuzzing (Xia et al., 2023a; Deng et al., 2023), test generation (Schäfer et al., 2023), automated program repair (Xia et al., 2023b), and source-level algorithmic optimization Madaan et al. (2023). The introduction of fill-in-the-middle capabilities is especially useful for software engineering use cases such as code completion, and has become common in recent code models such as InCoder (Fried et al., 2023), SantaCoder (Allal et al., 2023), StarCoder (Lozhkov et al., 2024), and Code Llama (Rozière et al., 2023). A large number of useful applications have been explored for LLMs, however, only very few are directly focused on compilation tasks.

Language models over IR. While LLMs have found broad adoption for coding tasks, few operate at the level of compilers. Gallagher et al. (2022) train a RoBERTA architecture on LLVM-IR for the purpose of code weakness identification, and Transcoder-IR (Szafraniec et al., 2022) uses LLVM-IR as a pivot point for source-to-source translation. Few LLMs include compiler IRs in their training, and of those that do, IRs comprise a tiny fraction of the data compared to other programming languages. StarCoder 2 Lozhkov et al. (2024) and DeepSeek-Coder Guo et al. (2024) include 7.7 GB (0.4%) and 0.91 GB (0.1%) of LLVM-IR respectively in their training data. LLM Compiler is pretrained on 422 GB of LLVM-IR, and additional LLVM-IR during fine-tuning, and assembly code which makes up at least 85% of the total training data.

Paul et al. (2024) create SLTrans, a 26 B token dataset which pairs high level source code with corresponding LLVM-IR. Like our dataset, they include different source languages and optimization levels for their IR, however, their optimization is limited to *-Oz* and *-O3*. They train IRCoder on 800 M tokens of SLTrans and demonstrate how it improves the code reasoning capabilities of underlying base models. IRCoder and StarCoder 2 present their models with LLVM-IR. We include both LLVM-IR as well as native assembly code from multiple source languages and for multiple architecture targets.

With the increasing interest in IR to improve the performance of code generation models, new datasets are emerging. For example, ComPile (Grossman et al., 2024), a 2.4 TB dataset of unoptimized LLVM-IR.

Machine Learning in Compilers. Many works have applied machine learning in compilers (Leather & Cummins, 2020; Ashouri et al., 2022; Cummins et al., 2017; Phothilimthana et al., 2021; Seeker et al., 2024). Compiler pass ordering has been exploited for decades. Over the years there have been several approaches using machine learning (Liang et al., 2023; Agakov et al., 2006; Ogilvie et al., 2017; Jayatilaka et al., 2021; Queiroz Jr & da Silva, 2023; Grubisic et al., 2024a). Neural machine translation is an emerging field that uses language models to transform code from one language to another. Prior examples include compiling C to assembly (Armengol-Estapé & O'Boyle, 2021), assembly to C (Armengol-Estapé et al., 2024; Hosseini & Dolan-Gavitt, 2022), and source-to-source (Lachaux et al., 2020).

7 Discussion

In this paper, we introduced LLM Compiler, a novel family of large language models specifically designed to address the challenges of code and compiler optimization. By extending the capabilities of the foundational Code Llama model, LLM Compiler provides a robust, pre-trained platform that significantly enhances the understanding and manipulation of compiler intermediate representations and assembly language.

We release LLM Compiler under a bespoke commercial license to facilitate widespread access and collaboration, enabling both academic researchers and industry practitioners to explore, modify, and extend the model according to their specific needs.

7.1 Limitations

We have shown that LLM Compiler performs well at compiler optimization tasks and has improved understanding of compiler representations and assembly code over prior works, but there are limitations. The main limitation is the finite sequence length of inputs (context window). LLM Compiler supports a 16k token context windows, but program codes may be far longer. For example, 67% of MiBench translation units exceeded this context window when formatted as flag tuning prompts, shown in Table 10. To mitigate this we split larger translation units into individual functions, though this limits the scope of optimization that can be performed, and still 18% of the split translation units remain too large for the model to accept as input. Researchers are adopting ever-increasing context windows (Ding et al., 2023), but finite context windows remain a common concern with LLMs.

A second limitation, common to all LLMs, is the accuracy of model outputs. Users of LLM Compiler are advised to assess their models using evaluation benchmarks specific to compilers. Given that compilers are not bug-free, any suggested compiler optimizations must be rigorously tested. When a model decompiles assembly code, its accuracy should be confirmed through round trip, manual inspection, or unit testing. For some applications LLM generations can be constrained to regular expressions (Grubisic et al., 2024b), or combined with automatic verification to ensure correctness (Taneja et al., 2024).

References

F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M.F.P. O'Boyle, J. Thomson, M. Toussaint, and C.K.I. Williams. Using machine learning to focus iterative optimization. In *CGO*, 2006.

Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, Logesh Kumar Umapathi, Carolyn Jane Anderson, Yangtian Zi, Joel Lamy Poirier, Hailey Schoelkopf, Sergey Troshin, Dmitry Abulkhanov, Manuel Romero, Michael Lappert, Francesco De Toni, Bernardo García del Río, Qian Liu, Shamik Bose, Urvashi Bhattacharyya, Terry Yue Zhuo, Ian Yu, Paulo Villegas, Marco Zocca, Sourab Mangrulkar, David Lansky, Huu Nguyen, Danish Contractor, Luis Villa, Jia Li, Dzmitry Bahdanau, Yacine Jernite, Sean Hughes, Daniel Fried, Arjun Guha, Harm de Vries, and Leandro von Werra. SantaCoder: Don't Reach for the Stars! *arXiv:2301.03988*, 2023.

Jordi Armengol-Estapé and Michael FP O'Boyle. Learning C to x86 Translation: An Experiment in Neural Compilation. *arXiv:2108.07639*, 2021.

Jordi Armengol-Estapé, Jackson Woodruff, Chris Cummins, and Michael FP O'Boyle. SLaDe: A Portable Small Language Model Decompiler for Optimized Assembler. In *CGO*, 2024.

Amir H Ashouri, Mostafa Elhoushi, Yuzhe Hua, Xiang Wang, Muhammad Asif Manzoor, Bryan Chan, and Yaoqing Gao. MLGOPerf: An ML Guided Inliner to Optimize Performance. *arXiv:2207.08389*, 2022.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program Synthesis with Large Language Models. *arXiv:2108.07732*, 2021.

Ying Cao, Ruigang Liang, Kai Chen, and Peiwei Hu. Boosting Neural Networks to Decompile Optimized Binaries. In *ACSAC*, 2022.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating Large Language Models Trained on Code. *arXiv:2107.03374*, 2021.

Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. End-to-End Deep Learning of Optimization Heuristics. In *PACT*, 2017.

Chris Cummins, Zacharias Fisches, Tal Ben-Nun, Torsten Hoefler, Michael O'Boyle, and Hugh Leather. ProGraML: A Graph-based Program Representation for Data Flow Analysis and Compiler Optimizations. In *ICML*, 2021.

Chris Cummins, Bram Wasti, Jiadong Guo, Brandon Cui, Jason Ansel, Sahir Gomez, Somya Jain, Jia Liu, Olivier Teytaud, Benoit Steiner, Yuandong Tian, and Hugh Leather. CompilerGym: Robust, Performant Compiler Optimization Environments for AI Research. In *CGO*, 2022.

Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. Large Language Models Are Zero-Shot Fuzzers: Fuzzing Deep-Learning Libraries via Large Language Models. In *ISSTA*, 2023.

Jiayu Ding, Shuming Ma, Li Dong, Xingxing Zhang, Shaohan Huang, Wenhui Wang, and Furu Wei. LongNet: Scaling Transformers to 1,000,000,000 Tokens. *arXiv:2307.02486*, 2023.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. CodeBERT: A Pre-trained Model for Programming and Natural Languages. *arXiv:2002.08155*, 2020.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. InCoder: A Generative Model for Code Infilling and Synthesis. *arXiv:2204.05999*, 2023.

G. G. Fursin, M. F. P. O'Boyle, and P. M. W. Knijnenburg. Evaluating Iterative Compilation. In *LCPC*, 2005.

Philip Gage. A New Algorithm for Data Compression. *C Users Journal*, 12(2), 1994.

Shannon K Gallagher, William E Klieber, and David Svoboda. LLVM Intermediate Representation for Code Weakness Identification, 2022.

Aiden Grossman, Ludger Paehler, Konstantinos Parasyris, Tal Ben-Nun, Jacob Hegna, William Moses, Jose M Monsalve Diaz, Mircea Trofin, and Johannes Doerfert. ComPile: A Large IR Dataset from Production Sources. *arXiv:2309.15432*, 2024.

Dejan Grubisic, Chris Cummins, Volker Seeker, and Hugh Leather. Compiler generated feedback for Large Language Models. *arXiv:2403.14714*, 2024a.

Dejan Grubisic, Chris Cummins, Volker Seeker, and Hugh Leather. Priority Sampling of Large Language Models for Compilers. *arXiv:2402.18734*, 2024b.

Yi Gui, Yao Wan, Hongyu Zhang, Huifang Huang, Yulei Sui, Guandong Xu, Zhiyuan Shao, and Hai Jin. Cross-language binary-source code matching with intermediate representations. In *SANER*, 2022.

Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, Adil Salim, Shital Shah, Harkirat Singh Behl, Xin Wang, Sébastien Bubeck, Ronen Eldan, Adam Tauman Kalai, Yin Tat Lee, and Yuanzhi Li. Textbooks Are All You Need. *arXiv:2306.11644*, 2023.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. DeepSeek-Coder: When the Large Language Model Meets Programming – The Rise of Code Intelligence. *arXiv:2401.14196*, 2024.

Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard B Brown. MiBench: A free, commercially representative embedded benchmark suite. In *WWC*. IEEE, 2001.

Iman Hosseini and Brendan Dolan-Gavitt. Beyond the C: Retargetable Decompilation using Neural Machine Translation. *arXiv:2212.08950*, 2022.

Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. Large Language Models for Software Engineering: A Systematic Literature Review. *arXiv:2308.10620*, 2023.

Tarindu Jayatilaka, Hideto Ueno, Giorgis Georgakoudis, EunJung Park, and Johannes Doerfert. Towards Compile-Time-Reducing Compiler Optimization Selection via Machine Learning. In *ICPP*, 2021.

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A Survey on Large Language Models for Code Generation. *arXiv:2406.00515*, 2024.

Marie-Anne Lachaux, Baptiste Roziere, Lowik Chanussot, and Guillaume Lample. Unsupervised Translation of Programming Languages. *arXiv:2006.03511*, 2020.

Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO*, 2004.

Hugh Leather and Chris Cummins. Machine Learning in Compilers: Past, Present and Future. In *FDL*, 2020.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with AlphaCode. *Science*, 378(6624), 2022.

Youwei Liang, Kevin Stone, Ali Shameli, Chris Cummins, Mostafa Elhoushi, Jiadong Guo, Benoit Steiner, Xiaomeng Yang, Pengtao Xie, Hugh Leather, and Yuandong Tian. Learning Compiler Pass Orders using Coreset and Normalized Value Prediction. In *ICML*, 2023.

Ilya Loshchilov and Frank Hutter. Decoupled Weight Decay Regularization. *arXiv:1711.05101*, 2017.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, et al. StarCoder 2 and The Stack v2: The Next Generation. *arXiv:2402.19173*, 2024.

Aman Madaan, Alexander Shypula, Uri Alon, Milad Hashemi, Parthasarathy Ranganathan, Yiming Yang, Graham Neubig, and Amir Yazdanbakhsh. Learning performance-improving code edits. *arXiv:2302.07867*, 2023.

Margaret Mitchell, Simone Wu, Andrew Zaldivar, Parker Barnes, Lucy Vasserman, Ben Hutchinson, Elena Spitzer, Inioluwa Deborah Raji, and Timnit Gebru. Model cards for model reporting. In *FAT*, pp. 220–229. ACM, 2019.

William F. Ogilvie, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. Minimizing the Cost of Iterative Compilation with Active Learning. In *CGO*, 2017.

OpenAI. GPT-4 Technical Report. *arXiv:2303.08774*, 2023.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. BLEU: A Method for Automatic Evaluation of Machine Translation. In *ACL*, 2002.

Indraneil Paul, Goran Glavaš, and Iryna Gurevych. IRCoder: Intermediate Representations Make Language Models Robust Multilingual Code Generators. *arXiv:2403.03894*, 2024.

Phitchaya Mangpo Phothilimthana, Amit Sabne, Nikhil Sarda, Karthik Srinivasa Murthy, Yanqi Zhou, Christof Angermueller, Mike Burrows, Sudip Roy, Ketan Mandke, Rezsa Farahani, et al. A Flexible Approach to Autotuning Multi-pass Machine Learning Compilers. In *PACT*, 2021.

LLVM PM. Using the New Pass Manager — LLVM 17.0.6 documentation, 2021. URL `https://llvm.org/docs/NewPassManager.html`.

Nilton Luiz Queiroz Jr and Anderson Faustino da Silva. A graph-based model for build optimization sequences: A study of optimization sequence length impacts on code size and speedup. *COLA*, 74, 2023.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code Llama: Open Foundation Models for Code. *arXiv:2308.12950*, 2023.

Eric Schulte, Jason Ruchti, Matt Noonan, David Ciarletta, and Alexey Loginov. Evolving Exact Decompilation. In *BAR*, 2018.

Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. Adaptive Test Generation Using a Large Language Model. *arXiv:2302.06527*, 2023.

Volker Seeker, Chris Cummins, Murray Cole, Björn Franke, Kim Hazelwood, and Hugh Leather. Revealing Compiler Heuristics Through Automated Discovery and Optimization. In *CGO*, 2024.

Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568, 2024.

Marc Szafraniec, Baptiste Roziere, Francois Charton, Hugh Leather, Patrick Labatut, and Gabriel Synnaeve. Code Translation with Compiler Representations. *arXiv:2207.03578*, 2022.

Jubi Taneja, Avery Laird, Cong Yan, Madan Musuvathi, and Shuvendu K. Lahiri. LLM-Vectorizer: LLM-based Verified Loop Vectorizer. *arXiv:2406.04693*, 2024.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv:2302.13971*, 2023a.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open Foundation and Fine-Tuned Chat Models. *arXiv:2307.09288*, 2023b.

Mircea Trofin, Yundi Qian, Eugene Brevdo, Zinan Lin, Krzysztof Choromanski, and David Li. MLGO: a Machine Learning Guided Compiler Optimizations Framework. *arXiv:2101.04808*, 2021.

Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922*, 2023.

Zheng Wang and Michael O'Boyle. Machine Learning in Compiler Optimisation. *arXiv:1805.03441*, 2018.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. Magicoder: Empowering Code Generation with OSS-Instruct. *arXiv:2312.02120*, 2024.

Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. Universal Fuzzing via Large Language Models. *arXiv:2308.04748*, 2023a.

Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. Automated Program Repair in the Era of Large Pre-Trained Language Models. In *ICSE*, 2023b.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. CodeGeeX: A Pre-Trained Model for Code Generation with Multilingual Evaluations on HumanEval-X. *arXiv:2303.17568*, 2023.

A Acknowledgements

We would like to express our gratitude to all the people who helped us carry out this project:

- Our product and program management team: Abhishek Charnalia, Claire Roberts
- Participants in the red teaming exercises: Shengye Wan, Aaron Grattafiori, Chris Rohlf, Joshua Saxe, Ivan Evtimov
- Our sanctions team: Olivia Scheiber, Samuel Selvan, Kyle Watson, Jenlain Scott, Margaret Warner
- Our marketing and comms team: Ana Paula Kirschner Mofarrej, Lisa Brown Jaloza, Nisha Deo, Uzma Saeed
- Our leadership team: Joelle Pinneau, Naila Murray, Kim Hazelwood, Joe Spisak, Ragavan Srinivasan
- All the members of the Code Llama team: Sida Wang, Jade Copet, Yossef Adi, Olivier Duchenne, Quentin Carbonneaux, Jonas Gehring, Nicolas Usunier, Fabian Gloeckle, Taco Cohen, Kunhao Zheng, Sten Sootla, Vegard Mella, Pierre Chambon, Vegard Mella, Wes BOUAZIZ.

B Prompts

B.1 Compiler emulation prompts

Prompt:

```
[INST] Give the LLVM-IR for the following code when optimized using opt -p 'module(default<Oz>)':

<code>; ModuleID = '<stdin>'
source_filename = "-"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"

; Function Attrs: minsize nounwind optsize uwtable
define dso_local i32 @add_two(i32 noundef %0, i32 noundef %1) #0 {
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  store i32 %0, ptr %3, align 4, !tbaa !5
  store i32 %1, ptr %4, align 4, !tbaa !5
  %5 = load i32, ptr %3, align 4, !tbaa !5
  %6 = load i32, ptr %4, align 4, !tbaa !5
  %7 = add nsw i32 %5, %6
  ret i32 %7
}

attributes #0 = { minsize nounwind optsize uwtable "min-legal-vector-width"="0" "no-trapping-math"="true"
 ↪ "stack-protector-buffer-size"="8" "target-cpu"="x86-64" "target-features"="+cmov,+cx8,+fxsr,+mmx,+sse
 ↪,+sse2,+x87" "tune-cpu"="generic" }

!llvm.module.flags = !{!0, !1, !2, !3}
!llvm.ident = !{!4}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{i32 8, !"PIC Level", i32 2}
!2 = !{i32 7, !"PIE Level", i32 2}
!3 = !{i32 7, !"uwtable", i32 2}
!4 = !{!"clang version 17.0.6 (git@github.com:fairinternal/CodeGen.git
 ↪b05db9bbf7a92019267416c1bb9996fe6134e3f1)"}
!5 = !{!6, !6, i64 0}
!6 = !{!"int", !7, i64 0}
!7 = !{!"omnipotent char", !8, i64 0}
!8 = !{!"Simple C/C++ TBAA"}
</code>

The input code has instruction count 8 and binary size 65 bytes. [/INST]
```

Answer:

```
The LLVM-IR will have instruction count 2 and binary sise 53 bytes:

<code>; ModuleID = '<stdin>'
source_filename = "-"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"

; Function Attrs: minsize mustprogress nofree norecurse nosync nounwind optsize willreturn memory(none)
 ↪uwtable
define dso_local i32 @add_two(i32 noundef %0, i32 noundef %1) local_unnamed_addr #0 {
  %3 = add nsw i32 %1, %0
  ret i32 %3
}

attributes #0 = { minsize mustprogress nofree norecurse nosync nounwind optsize willreturn memory(none)
 ↪uwtable "min-legal-vector-width"="0" "no-trapping-math"="true" "stack-protector-buffer-size"="8" "
 ↪target-cpu"="x86-64" "target-features"="+cmov,+cx8,+fxsr,+mmx,+sse,+sse2,+x87" "tune-cpu"="generic" }

!llvm.module.flags = !{!0, !1, !2, !3}
!llvm.ident = !{!4}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{i32 8, !"PIC Level", i32 2}
!2 = !{i32 7, !"PIE Level", i32 2}
!3 = !{i32 7, !"uwtable", i32 2}
!4 = !{!"clang version 17.0.6 (git@github.com:fairinternal/CodeGen.git
 ↪b05db9bbf7a92019267416c1bb9996fe6134e3f1)"}
</code>
```

Listing 2: Prompt used for $IR \rightarrow IR$ compiler emulation.

Prompt:

```
[INST] Give the assembly for the following code when optimized using opt -p 'module(default<Oz>)':

<code>; ModuleID = '<stdin>'
source_filename = "-"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"

; Function Attrs: minsize nounwind optsize uwtable
define dso_local i32 @add_two(i32 noundef %0, i32 noundef %1) #0 {
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  store i32 %0, ptr %3, align 4, !tbaa !5
  store i32 %1, ptr %4, align 4, !tbaa !5
  %5 = load i32, ptr %3, align 4, !tbaa !5
  %6 = load i32, ptr %4, align 4, !tbaa !5
  %7 = add nsw i32 %5, %6
  ret i32 %7
}

attributes #0 = { minsize nounwind optsize uwtable "min-legal-vector-width"="0" "no-trapping-math"="true"
 ↪ "stack-protector-buffer-size"="8" "target-cpu"="x86-64" "target-features"="+cmov,+cx8,+fxsr,+mmx,+sse
 ↪,+sse2,+x87" "tune-cpu"="generic" }

!llvm.module.flags = !{!0, !1, !2, !3}
!llvm.ident = !{!4}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{i32 8, !"PIC Level", i32 2}
!2 = !{i32 7, !"PIE Level", i32 2}
!3 = !{i32 7, !"uwtable", i32 2}
!4 = !{!"clang version 17.0.6 (git@github.com:fairinternal/CodeGen.git
 ↪b05db9bbf7a92019267416c1bb9996fe6134e3f1)"}
!5 = !{!6, !6, i64 0}
!6 = !{!"int", !7, i64 0}
!7 = !{!"omnipotent char", !8, i64 0}
!8 = !{!"Simple C/C++ TBAA"}
</code>

The input code has instruction count 8 and binary size 65 bytes. [/INST]
```

Answer:

```
The assembly will have instruction count 2 and binary sise 53 bytes:

<code>  .text
  .file "-"
  .globl  add_two                         # -- Begin function add_two
  .type add_two,@function
add_two:                                # @add_two
  .cfi_startproc
# %bb.0:
  movl  %esi, %eax
  addl  %edi, %eax
  retq
.Lfunc_end0:
  .size add_two, .Lfunc_end0-add_two
  .cfi_endproc
                                        # -- End function
  .ident  "clang version 17.0.6 (git@github.com:fairinternal/CodeGen.git
   ↪b05db9bbf7a92019267416c1bb9996fe6134e3f1)"
  .section  ".note.GNU-stack","",@progbits
  .addrsig
</code>
```

Listing 3: Prompt used for *IR* → *assembly* compiler emulation.

B.2 Flag tuning prompts

Prompt:

```
[INST] Tell me how to optimize this LLVM-IR for object file size:

<code>; ModuleID = ’<stdin>’
source_filename = "-"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"

; Function Attrs: minsize nounwind optsize uwtable
define dso_local i32 @add_two(i32 noundef %0, i32 noundef %1) #0 {
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  store i32 %0, ptr %3, align 4, !tbaa !5
  store i32 %1, ptr %4, align 4, !tbaa !5
  %5 = load i32, ptr %3, align 4, !tbaa !5
  %6 = load i32, ptr %4, align 4, !tbaa !5
  %7 = add nsw i32 %5, %6
  ret i32 %7
}

attributes #0 = { minsize nounwind optsize uwtable "min-legal-vector-width"="0" "no-trapping-math"="true"
 ↪ "stack-protector-buffer-size"="8" "target-cpu"="x86-64" "target-features"="+cmov,+cx8,+fxsr,+mmx,+sse
 ↪,+sse2,+x87" "tune-cpu"="generic" }

!llvm.module.flags = !{!0, !1, !2, !3}
!llvm.ident = !{!4}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{i32 8, !"PIC Level", i32 2}
!2 = !{i32 7, !"PIE Level", i32 2}
!3 = !{i32 7, !"uwtable", i32 2}
!4 = !{!"clang version 17.0.6 (git@github.com:fairinternal/CodeGen.git
 ↪b05db9bbf7a92019267416c1bb9996fe6134e3f1)"}
!5 = !{!6, !6, i64 0}
!6 = !{!"int", !7, i64 0}
!7 = !{!"omnipotent char", !8, i64 0}
!8 = !{!"Simple C/C++ TBAA"}</code> [/INST]
```

Answer:

```
The code has object file size of 65 which can be reduced to 53 by running:

‘opt -p ’module(default<Oz>)’‘

This will produce code:

<code>; ModuleID = ’<stdin>’
source_filename = "-"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"

; Function Attrs: minsize mustprogress nofree norecurse nosync nounwind optsize willreturn memory(none)
 ↪uwtable
define dso_local i32 @add_two(i32 noundef %0, i32 noundef %1) local_unnamed_addr #0 {
  %3 = add nsw i32 %1, %0
  ret i32 %3
}

attributes #0 = { minsize mustprogress nofree norecurse nosync nounwind optsize willreturn memory(none)
 ↪uwtable "min-legal-vector-width"="0" "no-trapping-math"="true" "stack-protector-buffer-size"="8" "
 ↪target-cpu"="x86-64" "target-features"="+cmov,+cx8,+fxsr,+mmx,+sse,+sse2,+x87" "tune-cpu"="generic" }

!llvm.module.flags = !{!0, !1, !2, !3}
!llvm.ident = !{!4}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{i32 8, !"PIC Level", i32 2}
!2 = !{i32 7, !"PIE Level", i32 2}
!3 = !{i32 7, !"uwtable", i32 2}
!4 = !{!"clang version 17.0.6 (git@github.com:fairinternal/CodeGen.git
 ↪b05db9bbf7a92019267416c1bb9996fe6134e3f1)"}</code>
```

Listing 4: Prompt used for flag tuning.

Prompt:

```
[INST] Tell me how to optimize this LLVM-IR for object file size:

<code>; ModuleID = '<stdin>'
source_filename = "-"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"

; Function Attrs: minsize mustprogress nofree norecurse nosync nounwind optsize willreturn memory(none)
 ↪uwtable
define dso_local i32 @add_two(i32 noundef %0, i32 noundef %1) local_unnamed_addr #0 {
  %3 = add nsw i32 %1, %0
  ret i32 %3
}

attributes #0 = { minsize mustprogress nofree norecurse nosync nounwind optsize willreturn memory(none)
 ↪uwtable "min-legal-vector-width"="0" "no-trapping-math"="true" "stack-protector-buffer-size"="8" "
 ↪target-cpu"="x86-64" "target-features"="+cmov,+cx8,+fxsr,+mmx,+sse,+sse2,+x87" "tune-cpu"="generic" }

!llvm.module.flags = !{!0, !1, !2, !3}
!llvm.ident = !{!4}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{i32 8, !"PIC Level", i32 2}
!2 = !{i32 7, !"PIE Level", i32 2}
!3 = !{i32 7, !"uwtable", i32 2}
!4 = !{!"clang version 17.0.6 (git@github.com:fairinternal/CodeGen.git
 ↪b05db9bbf7a92019267416c1bb9996fe6134e3f1)"}</code> [/INST]
```

Answer:

```
The code has object file size of 53 which cannot be reduced further
```

Listing 5: Prompt and answer format for flag tuning task when no improvement can be made.

```
Provide your answer as a list of command line options to opt version 17.0.6, using the format:

"$ opt -p '<passes>'"

Where <passes> is a list of passes for the new pass manager, e.g. "function(dce),module(default<Oz>),
 ↪function(load-store-vectorizer)".

Only include the passes list. Do not include file paths or other flags such as -o. Terminate the opt
 ↪command line options with a newline.

Then report the optimized code that will be produced, delimited by <code> and </code> tags.

Finally, report the binary size the code before and after optimization using the template:

"Before optimization: X bytes. After optimization: Y bytes."

Where X and Y are placeholders for integer binary sizes in bytes. Binary size is the summation of the .
 ↪text and .data segment sizes of the object file generated by `clang-17 output.bc -c`, as reported by
 ↪the `size` tool.

Include no other text in your response.
```

Listing 6: Additional suffix appended to flag tuning prompts when evaluating GPT-4 and Code Llama - Instruct. We found that this additional context helped improve the coherence of model responses.

B.3 Disassembly prompts

Prompt:

```
[INST] Disassemble this code to LLVM-IR:

<code>  .text
  .file "-"
  .globl  add_two                        # -- Begin function add_two
  .type add_two,@function
add_two:                                # @add_two
  .cfi_startproc
# %bb.0:
  movl  %edi, -4(%rsp)
  movl  %esi, -8(%rsp)
  movl  -4(%rsp), %eax
  addl  -8(%rsp), %eax
  retq
.Lfunc_end0:
  .size add_two, .Lfunc_end0-add_two
  .cfi_endproc
                                        # -- End function
  .ident  "clang version 17.0.6 (git@github.com:fairinternal/CodeGen.git
   ↪b05db9bbf7a92019267416c1bb9996fe6134e3f1)"
  .section  ".note.GNU-stack","",@progbits
  .addrsig
</code> [/INST]
```

Answer:

```
<code>; ModuleID = '<stdin>'
source_filename = "-"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"

; Function Attrs: minsize nounwind optsize uwtable
define dso_local i32 @add_two(i32 noundef %0, i32 noundef %1) #0 {
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  store i32 %0, ptr %3, align 4, !tbaa !5
  store i32 %1, ptr %4, align 4, !tbaa !5
  %5 = load i32, ptr %3, align 4, !tbaa !5
  %6 = load i32, ptr %4, align 4, !tbaa !5
  %7 = add nsw i32 %5, %6
  ret i32 %7
}

attributes #0 = { minsize nounwind optsize uwtable "min-legal-vector-width"="0" "no-trapping-math"="true"
 ↪ "stack-protector-buffer-size"="8" "target-cpu"="x86-64" "target-features"="+cmov,+cx8,+fxsr,+mmx,+sse
 ↪,+sse2,+x87" "tune-cpu"="generic" }

!llvm.module.flags = !{!0, !1, !2, !3}
!llvm.ident = !{!4}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{i32 8, !"PIC Level", i32 2}
!2 = !{i32 7, !"PIE Level", i32 2}
!3 = !{i32 7, !"uwtable", i32 2}
!4 = !{!"clang version 17.0.6 (git@github.com:fairinternal/CodeGen.git
 ↪b05db9bbf7a92019267416c1bb9996fe6134e3f1)"}
!5 = !{!6, !6, i64 0}
!6 = !{!"int", !7, i64 0}
!7 = !{!"omnipotent char", !8, i64 0}
!8 = !{!"Simple C/C++ TBAA"}
</code>
```

Listing 7: Prompt used for disassembly.

```
Use LLVM version 17.0.6. Provide the IR enclosed by <code> and </code> tags.

Include no other text.
```

Listing 8: Additional suffix appended to disassembly prompts when evaluating GPT-4 and Code Llama - Instruct. We found that this additional context helped improve the coherence of model responses.

C Optimization Passes

Table 9: List of *opt* 17.0.6 transformation passes used to generate data for code optimization self-training and flag-tuning tasks.

Pass Name	Level	Pass Name	Level	Pass Name	Level
O0	Module	no-op-cgscc	CGSCC	lcssa	Function
O1	Module	inline	CGSCC	loop-data-prefetch	Function
O2	Module	coro-split	CGSCC	loop-load-elim	Function
O3	Module	function-attrs	CGSCC	loop-fusion	Function
Os	Module	aa-eval	Function	loop-distribute	Function
Oz	Module	adce	Function	loop-versioning	Function
always-inline	Module	add-discriminators	Function	pa-eval	Function
attributor	Module	aggressive-instcombine	Function	place-safepoints	Function
annotation2metadata	Module	assume-builder	Function	reassociate	Function
openmp-opt	Module	assume-simplify	Function	redundant-dbg-inst-elim	Function
openmp-opt-postlink	Module	alignment-from-assumptions	Function	reg2mem	Function
called-value-propagation	Module	annotation-remarks	Function	scalarize-masked-mem-intrin	Function
canonicalize-aliases	Module	bdce	Function	scalarizer	Function
constmerge	Module	break-crit-edges	Function	separate-const-offset-from-gep	Function
coro-early	Module	callsite-splitting	Function	sccp	Function
coro-cleanup	Module	consthoist	Function	sink	Function
cross-dso-cfi	Module	count-visits	Function	slp-vectorizer	Function
deadargelim	Module	constraint-elimination	Function	slsr	Function
elim-avail-extern	Module	chr	Function	speculative-execution	Function
extract-blocks	Module	coro-elide	Function	strip-gc-relocates	Function
forceattrs	Module	correlated-propagation	Function	tailcallelim	Function
globalopt	Module	dce	Function	vector-combine	Function
globalsplit	Module	dfa-jump-threading	Function	tlshoist	Function
hotcoldsplit	Module	div-rem-pairs	Function	declare-to-assign	Function
inferattrs	Module	dse	Function	early-cse	Function
inliner-wrapper	Module	fix-irreducible	Function	ee-instrument	Function
inliner-wrapper-no-mandatory-first	Module	flattencfg	Function	hardware-loops	Function
iroutliner	Module	make-guards-explicit	Function	lower-matrix-intrinsics	Function
lower-global-dtors	Module	gvn-hoist	Function	loop-unroll	Function
lower-ifunc	Module	gvn-sink	Function	simplifycfg	Function
lowertypetests	Module	infer-address-spaces	Function	loop-vectorize	Function
mergefunc	Module	instcombine	Function	instcombine	Function
name-anon-globals	Module	instsimplify	Function	mldst-motion	Function
partial-inliner	Module	irce	Function	gvn	Function
recompute-globalsaa	Module	float2int	Function	sroa	Function
rel-lookup-table-converter	Module	libcalls-shrinkwrap	Function	loop-flatten	Loop
rewrite-statepoints-for-gc	Module	inject-tli-mappings	Function	loop-interchange	Loop
rewrite-symbols	Module	instnamer	Function	loop-unroll-and-jam	Loop
rpo-function-attrs	Module	lower-expect	Function	canon-freeze	Loop
scc-oz-module-inliner	Module	lower-guard-intrinsic	Function	loop-idiom	Loop
strip	Module	lower-constant-intrinsics	Function	loop-instsimplify	Loop
strip-dead-debug-info	Module	lower-widenable-condition	Function	loop-deletion	Loop
strip-dead-prototypes	Module	guard-widening	Function	loop-simplifycfg	Loop
strip-debug-declare	Module	load-store-vectorizer	Function	loop-reduce	Loop
strip-nondebug	Module	loop-simplify	Function	indvars	Loop
strip-nonlinetable-debuginfo	Module	loop-sink	Function	loop-unroll-full	Loop
synthetic-counts-propagation	Module	lowerswitch	Function	loop-predication	Loop
wholeprogramdevirt	Module	mem2reg	Function	guard-widening	Loop
module-inline	Module	memcpyopt	Function	loop-bound-split	Loop
pseudo-probe-update	Module	mergeicmps	Function	loop-reroll	Loop
globaldce	Module	mergereturn	Function	loop-versioning-licm	Loop
ipsccp	Module	move-auto-init	Function	simple-loop-unswitch	Loop
embed-bitcode	Module	nary-reassociate	Function	loop-rotate	Loop
argpromotion	CGSCC	newgvn	Function	licm	LoopMssa
attributor-cgscc	CGSCC	jump-threading	Function	lnicm	LoopMssa
openmp-opt-cgscc	CGSCC	partially-inline-libcalls	Function		

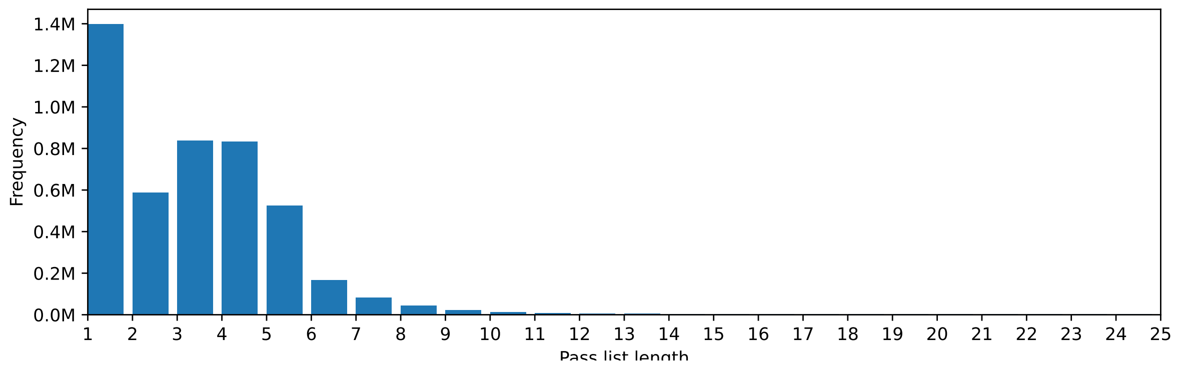


Figure 9: Length of autotuned pass lists.

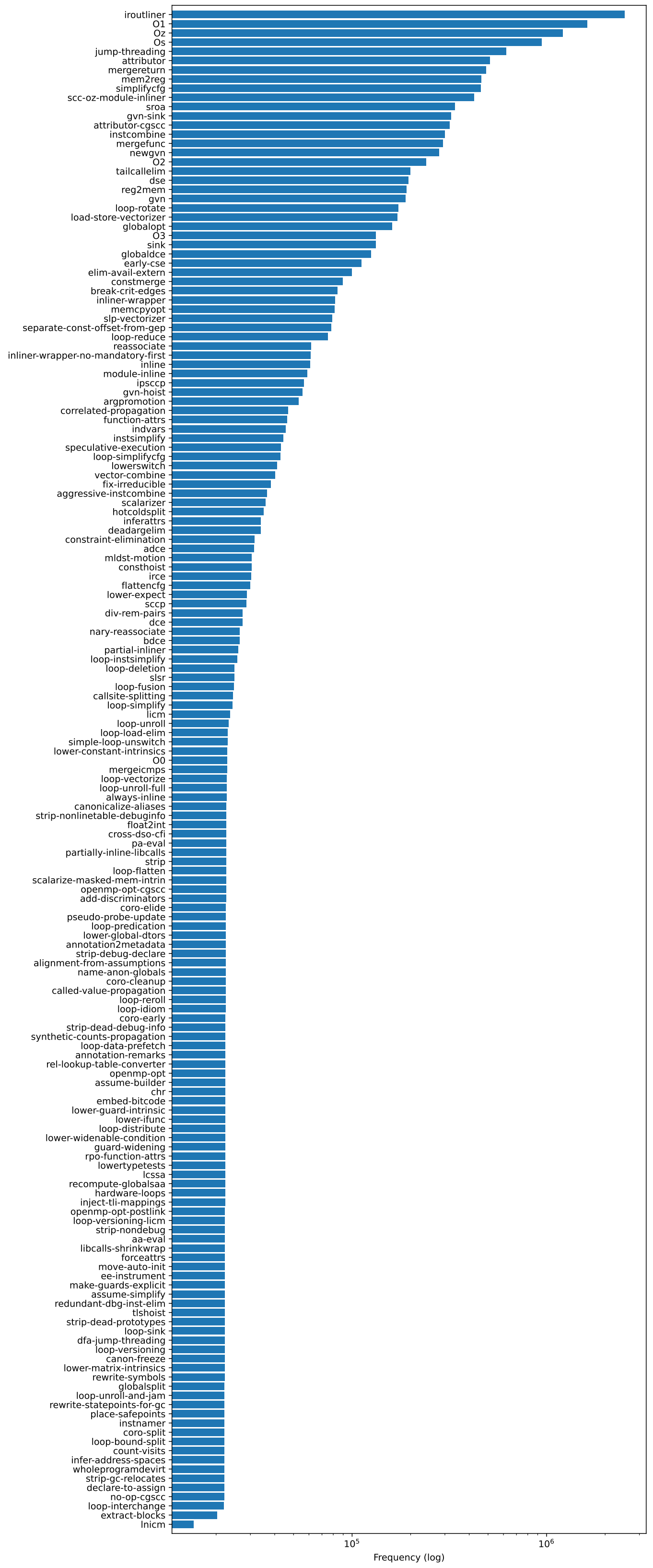


Figure 10: The frequency of passes in the best pass lists generated by the autotuner on our training programs.

D Benchmarks

Table 10: MiBench benchmarks used for flag tuning task evaluation.

		Without split		With split	
	Binary size	Translation units	Truncated prompts	Translation units	Truncated prompts
adpcm	816.7 kB	2		2	
basicmath	931.7 kB	4		4	
bitcount	821.1 kB	8		8	
blowfish	830.6 kB	7	3 (43%)	7	2 (29%)
crc32	818.4 kB	1		1	
dijkstra	946.0 kB	1		1	
fft	844.8 kB	3		3	
ghostscript	1.9 MB	296	222 (75%)	1,052	162 (15%)
gsm	58.8 kB	23	12 (52%)	37	10 (27%)
ispell	91.5 kB	12	8 (67%)	39	6 (15%)
jpeg_c	112.5 kB	54	39 (72%)	170	22 (13%)
jpeg_d	151.7 kB	54	39 (72%)	164	18 (11%)
lame	289.2 kB	32	22 (69%)	92	24 (26%)
patricia	949.3 kB	2	1 (50%)	3	
qsort	944.3 kB	1		1	
rsynth	151.4 kB	19	10 (53%)	27	3 (11%)
sha	5.3 kB	2	1 (50%)	3	
stringsearch	821.5 kB	4		4	
susan	911.4 kB	1	1 (100%)	13	7 (54%)
tiff2bw	442.1 kB	34	19 (56%)	134	24 (18%)
tiff2rgba	492.7 kB	34	19 (56%)	134	23 (17%)
tiffdither	441.2 kB	34	19 (56%)	133	23 (17%)
tiffmedian	453.0 kB	34	19 (56%)	139	26 (19%)
typeset	2.0 MB	51	43 (84%)	227	89 (39%)
Total		713	477 (67%)	2,398	439 (18%)

Table 11: MiBench benchmarks used for disassembly task evaluation.

	Translation units	Truncated prompts
adpcm	3	
basicmath	2	
bitcount	8	
blowfish	3	
crc32	1	
dijkstra	2	
fft	1	
ghostscript	1,264	2
gsm	35	
ispell	45	
jpeg_c	24	
jpeg_d	177	
lame	87	1
patricia	3	
qsort	1	
rsynth	33	1
sha	3	
stringsearch	5	
susan	7	
tiff2bw	3	
tiff2rgba	5	
tiffdither	2	
tiffmedian	158	
typeset	143	
Total	2015	4

E Model card

Model details	
Model Developers	Meta AI
Variations	LLM COMPILER comes in two model sizes: 7B and 13B parameters. Both variations have been trained on the same data. LLM COMPILER FTD, available in the same sizes, extends these with further training.
Input	Models input text only.
Output	Models output text only.
Model Architecture	LLM COMPILER and its variants are autoregressive language models using optimized transformer architectures. All models were fine-tuned with up to 16K tokens.
Model Dates	LLM COMPILER and its variants have been trained between January and May 2024.
Status	This is a static model trained on an offline dataset.
Licence	A custom commercial license is available at: `ai.meta.com/resources/models-and-libraries/llama-downloads/`.
Where to send comments	Instructions on how to provide feedback or comments on the model can be found in the model README, or by opening an issue in the GitHub repository (`https://github.com/facebookresearch/llmcompiler/`).
Intended Use	
Intended Use Cases	LLM COMPILER and its variants are intended for commercial and research use in English and relevant programming languages. The foundation model LLM COMPILER can be adapted for a variety of code optimization and understanding tasks.
Out-of-Scope Uses	Use in any manner that violates applicable laws or regulations (including trade compliance laws). Use in languages other than English. Use in any other way that is prohibited by the Acceptable Use Policy and Licensing Agreement for LLM COMPILER and its variants.
Hardware and Software	
Training Factors	We used custom training libraries. The training and fine-tuning of the released models have been performed on Meta's Research Super Cluster.
Carbon Footprint	In aggregate, training all 4 LLM COMPILER models required 264K GPU hours of computation on hardware of type A100-80GB (TDP of 350-400W). Estimated total emissions were 64.12 tCO2eq, 100% of which were offset by Meta's sustainability program.
Training Data	
All experiments reported here and the released models have been trained and fine-tuned using the same data as CODE LLAMA with different weights (see Section 2 and Table 1).	
Evaluation Results	
See evaluations for the main models and detailed ablations Section 5.	
Ethical Considerations and Limitations	
LLM COMPILER and its variants are a new technology that carries risks with use. Testing conducted to date has been in English, and has not covered, nor could it cover all scenarios. For these reasons, as with all LLMs, LLM COMPILER 's potential outputs cannot be predicted in advance, and the model may in some instances produce inaccurate or objectionable responses to user prompts. Therefore, before deploying any applications of LLM COMPILER, developers should perform safety testing and tuning tailored to their specific applications of the model. Please see the Responsible Use Guide available available at `https://ai.meta.com/llama/responsible-user-guide`.	

Table 12: Model card (Mitchell et al., 2019) for LLM COMPILER and LLM COMPILER FTD.