Large Language Models for Compiler Optimization

***Abstract*—We explore the novel application of Large Language Models to code optimization. We present a 7B-parameter transformer model trained from scratch to optimize LLVM assembly for code size. The model takes as input unoptimized assembly and outputs a list of compiler options to best optimize the program. Crucially, during training, we ask the model to predict the instruction counts before and after optimization, and the optimized code itself. These auxiliary learning tasks significantly improve the optimization performance of the model and improve the model's depth of understanding.**

We evaluate on a large suite of test programs. Our approach achieves a 3.0% improvement in reducing instruction counts over the compiler, outperforming two state-of-the-art baselines that require thousands of compilations. Furthermore, the model shows surprisingly strong code reasoning abilities, generating compilable code 91% of the time and perfectly emulating the output of the compiler 70% of the time.

I. Introduction

There is increasing interest in Large Language Models (LLMs) for software engineering domains such as code generation [1–9], code translation [10–12], and code testing [13–15]. Models such as Code Llama [9], Codex [8], and ChatGPT [16] have a good statistical understanding of code and suggest likely completions for unfinished code, making them useful for editing and creating software. However, it appears they have not been trained specifically to optimize code. ChatGPT, for instance, will 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 it easily gets confused and makes mistakes, frequently resulting in incorrect code.

Prior works on machine learning-guided code optimization have used hand-built features [17–19], all the way to graph neural networks (GNNs) [20, 21]. 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, MLGO [17] uses numeric features to provide hints for function inlining, but cannot faithfully reproduce the call graph or control flow, etc. ProgramL [21] forms graphs of the program to pass to a GNN, but it excludes the values for constants and some type information which prevents reproducing instructions with fidelity.

In this work, we ask: can Large Language Models learn to optimize code? 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 universal, portable, and accessible interface, and unlike prior approaches is not specialized to any particular task.

We started our investigation into the code-optimizing power of LLMs by replicating the optimizing transformations present in compilers, targeting the industry standard LLVM [22] compiler. LLVM's optimizer is extremely complex and contains thousands of rules, algorithms, and heuristics in over 1M lines of C++ code. Our expectation was that while LLMs have shown great progress in natural language translation and code generation tasks, they would be incapable of emulating such a complex system. Understanding and applying compiler optimizations require multiple levels of reasoning, arithmetic computation capabilities, and applying complex data structure and graph algorithms, which are capabilities LLMs have shown to lack [23, 24].

We thought this would be a paper about the obvious failings of LLMs that would serve as motivation for future clever ideas to overcome those failings. We were entirely taken by surprise to find that in many cases a sufficiently trained LLM can not only predict the best optimizations to apply to an input code, but it can also directly perform the optimizations without resorting to the compiler at all!

Our approach is simple. We begin with a 7B-parameter LLM architecture, taken from LLaMa 2 [25], and initialize it from scratch. We then train it on millions of examples of LLVM assembly, coupled with the best compiler options found by a search for each assembly, as well as the resulting assembly from performing those optimizations. From these examples alone the model learns to optimize code with remarkable accuracy.

Our singular contribution is the first application of LLMs to optimizing code. We construct LLMs solely for the purpose of compiler optimization and show that they achieve a single-compile 3.0% improvement in code size reduction over the compiler versus a search-based approach which achieves 5.0% with $2.5e^9$ compilations and versus state of the state-of-the-art ML approaches that cause regressions and require thousands of compilations. We provide auxiliary experiments and code examples to further characterize the potential and limits of LLMs for code reasoning. Overall we find their efficacy remarkable and think that these results will be of interest to the community.

II. Pass Ordering with LLMs

In this work we target compiler pass ordering. The pass ordering task is to select from the set of optimizing transformation passes available in a compiler the list of passes that will produce the best result for a particular input code. Manipulating pass orders has been shown to have a considerable impact on both runtime performance and code size [19, 26].

Machine learning approaches to this task have shown good results previously, but struggle with generalizing across different programs [27]. 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, making

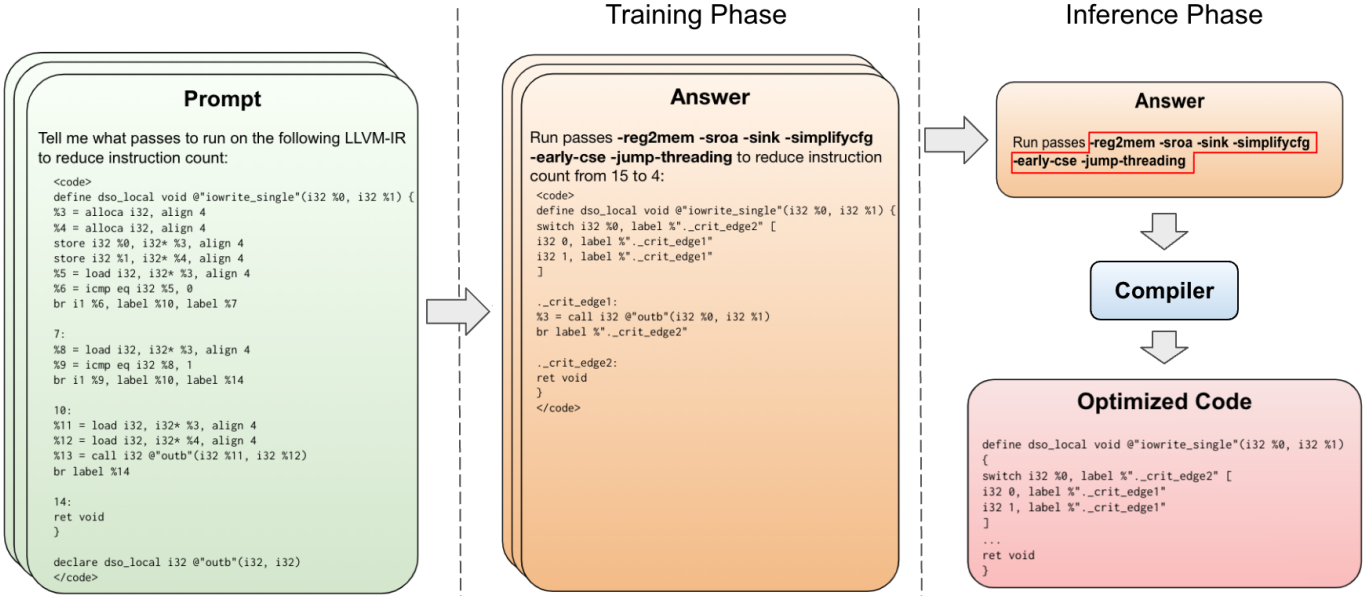


Figure 1: Overview of our approach, showing the model input (Prompt) and output (Answer) during training and inference. The prompt contains unoptimized code. The answer contains an optimization pass list, instruction counts, and the optimized code. During inference we generate only the optimization pass list which we feed into the compiler, ensuring that the optimized code is correct.

Table I: Training data. Each LLVM-IR function is autotuned and used to create a (Prompt, Answer) pair. The n tokens column shows the number of tokens when the prompt is encoded using the Llama 2 [25] tokenizer.

	n functions	unoptimized instruction count	size on disk	n tokens
Handwritten	610,610	8,417,799	653.5 MB	214,746,711
Synthetic	389,390	13,775,149	352.3 MB	158,435,151
Total	1,000,000	16,411,249	1.0 GB	373,181,862

Table II: Test data.

	n functions	unoptimized instruction count	-Oz instruction count
AI-SOCO [31]	8,929	97,800	47,578
ExeBench [32]	26,806	386,878	181,277
POJ-104 [33]	310	8,912	4,492
Transcoder [12]	17,392	289,689	129,611
CSmith [34]	33,794	647,815	138,276
YARPGen [35]	12,769	285,360	144,539
Total	100,000	1,716,354	645,773

them impractical for real-world use. We hypothesized that a large language model with sufficient reasoning power would be able to learn to make good optimization decisions without needing this.

Most prior work on LLMs for code operates on source languages such as Python. Instead, for the pass ordering problem we require reasoning at the lower level of compiler assembly, known as the Intermediate Representation (IR). While there exist curated datasets of source languages for pretraining LLMs (e.g. [28–30]), compiler IRs do not make up a significant portion of these datasets, and though models like ChatGPT show some promise of understanding, their ability to reason about IR is far inferior to source languages.

We target optimizing LLVM pass orders for code size as in prior works [17, 27], using IR instruction count as an (imperfect) proxy for binary size. The 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.

A. *Prompts*

We present the model with an unoptimized LLVM-IR (such as emitted by the *clang* frontend) and ask it to produce a list of optimization passes that should be applied to it. Figure 1 shows the format of the input prompt and output text.

In this work, we target LLVM 10 and use the optimization flags from `opt`. There are 122 optimization passes to choose from and passes can be selected more than once in a single sequence. We also include the 6 meta-flags (-O0, -O1, -O2, -O3, -Oz, and -Os) that may each occur only once per pass list. Pass lists can be any length, though in our experiments we found typically up to 9 passes long, for a combinatorial search space of around 10^{18}.

As shown in Figure 1, we also include two auxiliary tasks: i) generating the instruction counts of the code before and after the optimizations are applied and ii) generating the output IR after the optimizations are applied. We hypothesize that these would enable better pass-ordering decisions by forcing a deep understanding of the mechanics of code optimization.

We verify this experimentally in Section V-B.

While the model is trained to generate instruction counts and optimized IR, we do not need those auxiliary tasks for deployment. All we need to do is generate the pass list which we then execute using the compiler. We thus sidestep the problems of correctness that plague techniques that require the output of the model to be trustworthy [10–12, 36].

B. LLVM-IR Normalization

We normalize the LLVM-IR that is used for training the LLM using the following rules: we discard comments, debug metadata and attributes, and ensure consistent whitespace by feeding the IR through a custom lexer that retains newlines but standardizes other whitespace and strips indentation. We do this to reduce the length of the LLVM-IR to make maximum use of the limited input size of the LLM (Section III-A). The code in Figure 1 has been processed in this manner.

III. The Model

We use the ubiquitous transformer architecture [37]. The transformer is an artificial neural network that employs self-attention over a fixed-size context window.

The input text is first tokenized into words and subword units. These are embedded into continuous vector representations and provided as input to the transformer's encoder, where self-attention mechanisms capture contextual relationships between tokens to encourage the model to understand and process the input text's semantic structure.

The output text is produced by iteratively generating one token at a time. The decoder takes the encoded input along with any previously generated tokens and uses self-attention to predict the next token in the sequence. We greedily sample during decoding to select the most likely token sequence. This process continues until an end-of-sequence token is generated or a predefined maximum length is reached.

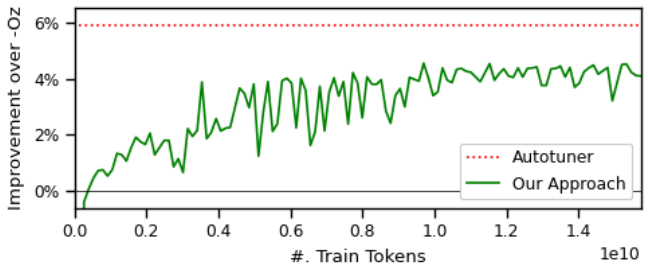
A. Model Architecture

We use the same model architecture and Byte Pair Encoding (BPE) [38] tokenizer as Llama 2 [25], but train our model from scratch. We use the smallest of the Llama 2 configurations: 32 attention heads, 4,096 hidden dimensions, and 32 layers, for a total of 7B parameters.

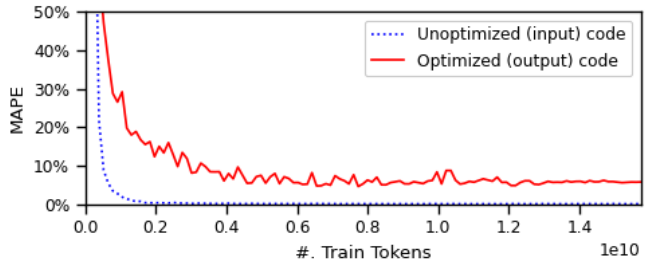
The maximum length of a (prompt, answer) pair is defined by the sequence length. In this work, we use a sequence length of 2,048 tokens. The Llama 2 tokenizer achieves an average of 2.02 characters per token when encoding LLVM-IR, so this provides an approximate upper limit on the longest LLVM-IR we can train on at 2KB (since 2KB prompt and 2KB answer $\approx$ 2,048 tokens).

B. Training Data

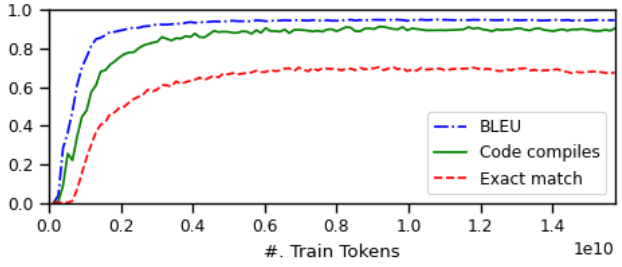
We assembled a large corpus of unoptimized LLVM-IR functions, summarized in Table I. We extracted the functions from datasets of publicly available handwritten C/C++ code and supplemented this with synthetic code generated by C/C++ compiler test generators. In total, our training corpus comprises



(a) Performance of generated pass lists.



(b) Accuracy at predicting instruction counts.



(c) Model-optimized code metrics.

Figure 2: Performance on holdout validation set during training. We evaluate performance every 250 training steps (131M train tokens). Parity with -Oz is reached at 393M tokens and peak performance at 10.9B tokens.

1,000,000 deduplicated IR functions, totaling 373M training tokens. We operate at the level of individual IR functions rather than entire modules to maximize the amount of data we can fit inside a 2,048-token sequence length.

To find the list of optimization passes that will produce the smallest instruction count we employ *autotuning*. Our autotuner combines random search and all-to-all results broadcasting between functions, inspired by the work of Liang et. al. [20]. For each function we run random search for a fixed amount of time (780 seconds) and then minimize the best pass list by iteratively removing individual randomly chosen passes to see if they contribute to the instruction count. If not, they are discarded. After performing this on each of the functions we aggregate the set of unique best pass lists and broadcast them across all other functions. Thus, if a pass list was found to work well on one function it is tried on all others.

In total, the autotuner compiled each training program an average of 37,424 times, achieving a 5.8% improvement in instruction count reduction over the baseline fixed pass ordering in the compiler provided by -Oz. For our purposes, this

Table III: Performance of different approaches to pass ordering on a test set of unseen LLVM-IR functions from Table II. All metrics are *w.r.t.* -Oz. *Instructions saved* is summed over *functions improved* and *instructions regressed* is summed over *functions regressed*. *Overall improvement* is the sum total instruction count savings *w.r.t* -Oz. The Autotuner achieves the best performance but requires 2.5B additional compilations (949 CPU days). Our approach achieves 60% of the gains of the autotuner without invoking the compiler once.

	additional compilations	functions improved	functions regressed	instructions saved	instructions regressed	overall improvement
Autotuner	2,522,253,069	6,764	0	30,948	0	5.03%
AutoPhase [39]	4,500,000	1,558	8,400	6,522	32,357	-3.85%
Coreset-NVP [20]	442,747	3,985	6,072	16,064	28,405	-1.88%
Our Approach	0	4,136	526	21,935	3,095	3.01%

Table IV: Extending the models in Table III with "-Oz backup". If a model predicts a pass list *other than* -Oz, it also evaluates -Oz and selects the best. This prevents regressions *w.r.t* -Oz at the expense of additional compilations.

	additional compilations	overall improvement
AutoPhase [39]	4,600,000	1.02%
Coreset-NVP [20]	542,747	2.55%
Our Approach	5,721	3.52%

autotuning serves as a gold standard for the optimization of each function. While the instruction count savings discovered by the autotuner are significant, the computational cost to reach these wins was 9,016 CPU days. The goal of this work is to achieve some fraction of the performance of the autotuner using a predictive model that does not require running the compiler thousands of times.

C. Training

Starting from randomly initialized weights, we trained the model for 30,000 steps on 64 V100s for a total training time of 620 GPU days. We use the AdamW optimizer [40] with β_1 and β_2 values of 0.9 and 0.95. We use a cosine learning rate schedule with 1,000 warm-up steps, a peak learning rate of $1e{-}5$, and a final learning rate of 1/10th of the peak. We used a batch size of 256 and each batch contains 524,288 tokens for a total of 15.7B training tokens. The full 30,000 steps of training is 7.7 epochs (iterations over the training corpus).

During training, we evaluated the model on a holdout validation set of 1,000 unseen IRs that were processed in the same manner as the training set. We evaluate every 250 steps.

IV. Evaluation

In this section, we evaluate the ability of the model to generate pass lists for unseen code and to correctly perform optimization.

A. Training Results

Figure 2 shows the performance during training when evaluated on a holdout validation set of 1,000 unseen LLVM-IR functions. Peak validation performance was achieved by the model at 10.9B training tokens.

At peak performance, the code optimized using model-generated pass sequences contains 4.4% fewer instructions than when optimized using the compiler's built-in pass ordering (-Oz). The autotuner achieves a greater instruction count reduction of 5.6%, but this required 27 million compilations of the validation set. The model makes its predictions without invoking the compiler once.

Figure 2b shows the error of predicted input and output instruction counts. Prediction of instruction counts for unoptimized code rapidly approaches near-perfect accuracy. Prediction of output instruction count proves more challenging, reaching a Mean Average Percentage Error (MAPE) of 5.9%.

Figure 2c evaluates the quality of the generated code using three metrics. The *BLEU* [41] score shows the similarity between the model-generated code and a reference ground-truth code produced by the compiler using the generated pass list. *Code compiles* is the frequency that model-generated code compiles without error. *Exact match* tracks the frequency that the model-generated code is a character-by-character match of the compiler-generated code when optimized using the generated pass list (i.e. how many times BLEU=1).

At peak performance, the model achieves an impressive 90.5% rate of generating code that compiles without errors. Furthermore, a BLEU score of 0.952 shows that the model-optimized code closely approximates that of the compiler, and the exact match frequency is 70%. For comparison, a baseline that simply copies the unoptimized code to the output would achieve a BLEU score of 0.531 and an exact match frequency of 0%, demonstrating that significant manipulation of the input code is required to achieve such high scores.

By the end of training performance on the validation set had plateaued. We use the best-performing checkpoint and switch to a $100\times$ larger-scale evaluation for the remainder of the evaluation.

B. Comparison to State-of-the-Art

In this experiment, we perform a large-scale evaluation of the LLM's ability to predict pass lists in comparison to baselines.

Datasets We aggregate a broad suite of benchmark datasets for evaluation, summarized in Table II. We deduplicate and exclude IR functions identical to those we trained on. Our test data comprises code from a variety of domains including coding competitions (AI-SOCO [31], POJ-104 [33]), compiler test case generators (CSmith [34], YARPGen [35]), and miscellaneous publicly available code (ExeBench [32], Transcoder [12]).

Baselines We compare our approach to three baselines: AutoPhase [39], Coreset-NVP [20], and the Autotuner.

AutoPhase [39] is a reinforcement learning approach in which an agent is trained using Proximal Policy Optimization [42] to select the sequence of optimization passes that will

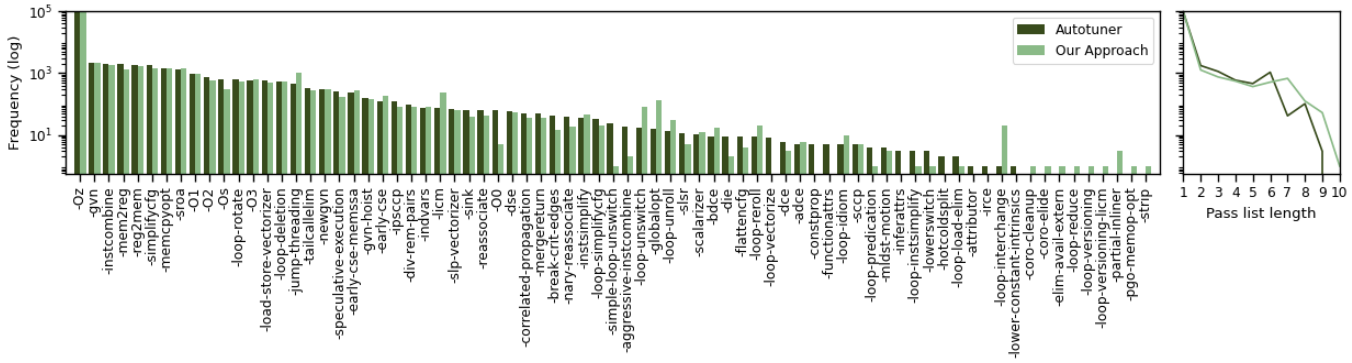


Figure 3: Frequency that passes occur in the pass list for each of the 100,000 test programs (left), and the length of pass lists (right). -Oz is the starting point for the autotuner and is the dominant result, being the best-found result for 93.2% of autotuned test programs and appearing in an additional 0.6% of pass lists as an additional flag. The model-generated pass distribution tracks the autotuner but slightly overpredicts -Oz (94.3%) and includes 9 passes that the autotuner used on the training set but not on the test set. Results are ordered by decreasing autotuner frequency.

```
define i32 @f1(i8 %0) {
 %2 = alloca i32, align 4
 %3 = alloca i8, align 1
 store i8 %0, i8* %3, align 1
 %4 = load i8, i8* %3, align 1
 %5 = zext i8 %4 to i32
 %6 = icmp sge i32 %5, 65
 br i1 %6, label %7, label %15

7:
 %8 = load i8, i8* %3, align 1
 %9 = zext i8 %8 to i32
 %10 = icmp sle i32 %9, 90
 br i1 %10, label %11, label %15

11:
 %12 = load i8, i8* %3, align 1
 %13 = zext i8 %12 to i32

<snip 21 lines...>

33:
 %34 = load i32, i32* %2, align 4
 ret i32 %34
}
```

(a) Input code (39 instructions).

```
define i32 @f1(i8 %0) {
 %2 = zext i8 %0 to i32
 %.off = add i8 %0, 191
 %3 = icmp ult i8 %.off, 26
 br i1 %3, label %4, label %6

4:
 %5 = add nsw i32 %2, 191
 br label %10

6:
 %.reload16.off = add nsw i32 %2, 159
 %7 = icmp ult i32 %.reload16.off, 26
 br i1 %7, label %10, label %8

8:
 %9 = icmp eq i8 %0, 32
 %. = select i1 %9, i32 26, i32 1
 br label %10

10:
 %.0.reg2mem.0 = phi i32 [%5, %4],
  [%., %8], [%.reload16.off, %6]
 ret i32 %.0.reg2mem.0
}
```

(b) Autotuned code (14 instructions) using passes: `-reg2mem -instcombine -Os -O1`.

```
define i32 @f1(i8 %0) {
 %2 = zext i8 %0 to i32
 %.off = add i8 %0, 191
 %3 = icmp ult i8 %.off, 26
 br i1 %3, label %6, label %._crit_edge

._crit_edge:
 %.off24 = add i8 %0, 159
 %4 = icmp ult i8 %.off24, 26
 br i1 %4, label %6, label %._crit_edge9

._crit_edge9:
 %5 = icmp eq i8 %0, 32
 %spec.select = select i1 %5,
  i32 26, i32 1
 ret i32 %spec.select

6:
 %.sink = phi i32 [191, %1],
  [159, %._crit_edge]
 %7 = add nsw i32 %.sink, %2
 ret i32 %7
}
```

(c) Model-optimized code (13 instructions) and pass list: `-reg2mem -simplifycfg -mem2reg -jump-threading -Os`.

Listing 1: An example IR function where the model suggests a better pass list than the autotuner, despite having never seen this code before. For this function the autotuner tried 26k different pass orderings. The pass list generated by the model appears 5 times in the training set of 1,000,000 examples.

maximize cumulative instruction count savings over a fixed-length episode. At each step, the program being optimized is represented to the agent as a 56-dimensional vector of instruction counts and other properties. We replicate the environment of [39] but use the implementation and expanded training regime from [27] in which the agent is trained for 100,000 episodes. We train the agent on the same data as our language model (Table I) and evaluate agent performance periodically during training on a holdout validation set. As in prior works, we use an action space and episode length of 45.

Coreset-NVP [20] is a technique that combines iterative search with a learned cost model. First, a greedy search is run on 17,500 benchmarks to determine a *Core set* of best pass lists. Then a *Neural Value Prediction* (NVP) is trained on the results of this search, using ProGraML [21] graphs processed by a Graph Convolutional Network as program representation. At inference, Coreset-NVP predicts the normalized reward and tries the first few pass sequences with the highest normalized reward. The total number of passes it is allowed to try for each benchmark is 45, following prior works. We use author-provided model weights to perform inference on our test set.

Finally, we compare it to the Autotuner that we used to generate training data. We autotuned the test dataset in the same manner as the training data, described in Section III-B.

Results Table III summarizes the results. Our approach outperforms -Oz, AutoPhase, and Coreset-NVP across all datasets. Overall, the thousands of optimization attempts that are afforded to the autotuner enable it to discover the best-performing pass lists.

AutoPhase and Coreset-NVP are both able to identify pass

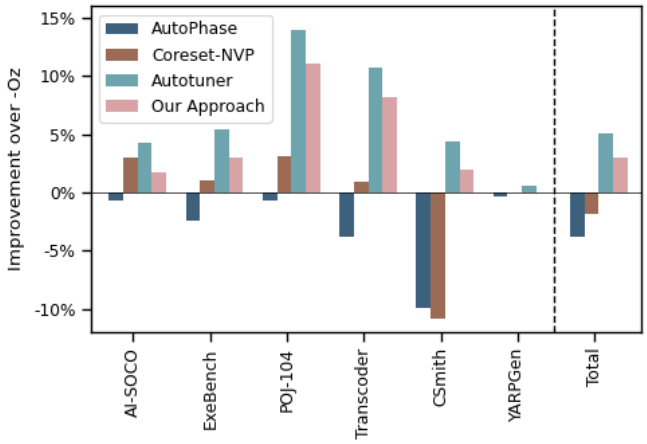


Figure 4: Improvement over -Oz by dataset. Handwritten code optimizes more.

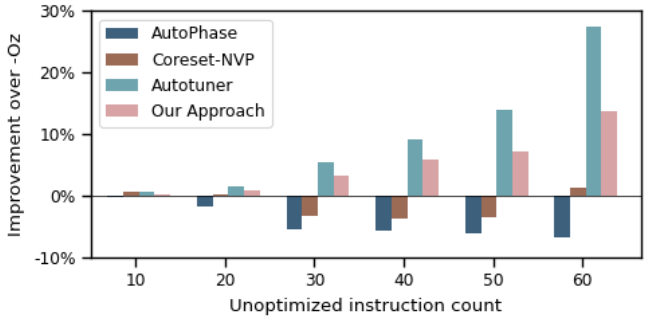


Figure 5: Improvement over -Oz by input size. Larger codes optimize more.

Table V: Compiler errors of model-optimized code on 100,000 unseen inputs.

error category	n
type error	5,777
instruction forward referenced	1,521
undefined value	1,113
invalid redefinition	616
syntax error	280
invalid value for constant	144
undefined function	112
index error	98
other	83
Total	9,744

lists that outperform -Oz but have an overall net negative impact on instruction count due to a large number of regressions. We propose a simple "-Oz backup" extension to overcome this: if a model predicts a pass list *other than* -Oz, we also run -Oz and select the best of the two options. This prevents regressions *w.r.t.* -Oz, but increases the number of additional compilations by the number of times the model predicts a pass list other than -Oz. Table IV shows the results of the techniques when evaluated in this manner. While this does not help the models find further improvements, the lack of regressions means that AutoPhase and Coreset-NVP now achieve overall improvements over -Oz, though still less than the LLM with or without the -Oz backup.

C. Evaluation of Generated Pass Lists

Figure 3 shows the frequency with which passes are selected by the autotuner and our model from the previous experiment. The distribution of passes selected by the model broadly tracks the autotuner. -Oz is the most frequently optimal pass. Excluding -Oz, model-generated pass lists have an average length of 3.4 (max 10), and autotuner pass lists have an average length of 3.1 (max 9). 105 of the pass lists generated by the model never appear in the training data.

In 710 cases the model-generated pass lists outperform the autotuner on the test set, though improvements are typically small. Listing 1 shows an example where the model-generated pass list simplifies control flow to fewer blocks, saving one further instruction.

Figure 4 breaks down the improvement of each approach to pass ordering by benchmark dataset. The biggest improvements over -Oz is found in the POJ-104 and Transcoder datasets, which both aggregate large amounts of handwritten code, while YARPGen, a random program generator for testing compilers, has the fewest opportunities for improving over -Oz.

We discovered that there is a strong correlation between the input program size and the potential performance improvement over -Oz that is found by both the autotuner and the model. Figure 5 plots this trend, showing clearly that larger programs have more opportunities to improve over -Oz.

D. Evaluation of Generated Code

In this section, we evaluate the quality of model-generated code. To do this we ran the auxiliary training task of generating optimized code for all 100k functions in the test set. Note that this is not required to generate the pass lists evaluated in the previous section. We have made minor edits to the code samples in this section for brevity such as omitting superfluous statements and shortening identifier names.

In 90.3% of cases, the model-generated optimized IR compiles, and in 68.4% of cases the output IR matches character-for-character the ground truth generated by the compiler. We taxonomize the different classes of errors for the 9.7% of cases where the generated IR does not compile in Table V, and Listing 2 provides code examples.

Most challenging to evaluate are the 21.9% of cases where the model-optimized code compiles but is not a character-by-character match with the compiler. There are two challenges: the first is that text precision metrics such as BLEU score are sensitive to differences in the code such as variable names and commutative operand order that do not affect the behavior of the code. Tools like LLVM-Canon [43] can help here but come with their own set of drawbacks. However, in many cases, it is unclear whether the behavior of two IRs is the same, so the second challenge we face is in evaluating *semantic* equivalency. Since not all of the datasets we use for testing provide driver scripts and input datasets for their code,

```
error: '%15' defined with type 'i32' but expected 'i1'
%or.cond = or i1 %14, %15
```

(a) The model defined `%15` as an integer but later tried to use it as a bool *(type error)*.

```
error: constant expression type mismatch
@.str = private unnamed_addr constant [493 x i8]
    c"<snip 492 chars ...>", align 1
```

(b) The model omitted a single character when transcribing a 493-character string-literal from the input code *(type error)*.

```
error: floating point constant invalid for type
%1 = tail call i32 @f1(float -0.47799998483256463,
                       float -1.8159999847412109)
```

(c) LLVM requires exact decimal values for floating-point constants. These model-generated values have repeating decimals in binary so are rejected *(invalid value for constant)*.

Listing 2: Compiler errors in model-optimized code.

```
define hidden signext i8 @f1() #0 {
  %1 = alloca i64, align 8
  store i64 3718042838174166437, i64* %1, align 8
  %2 = load i64, i64* %1, align 8
  %3 = trunc i64 %2 to i8
  ret i8 %3
}
```

(a) Input unoptimized code.

```
define hidden signext i8
@f1() #0 {
  ret i8 165
}
```

(b) Desired optimized code.

```
define hidden signext i8
@f1() #0 {
  ret i8 1
}
```

(c) Model-generated code.

Listing 3: An example where the model generates compilable code but fails to compute the correct answer for a numeric expression. Producing the correct result for this expression requires non-trivial mathematical reasoning.

we cannot use execution-based equivalence checks such as differential testing [44].

Listing 3 shows an example of model-generated code that has incorrect program semantics. Here, the lower 8 bits of a 64-bit literal are truncated and returned. The compiler performs this calculation and substitutes the correct value. The model recognizes that the expression can be calculated at compile time but fails to compute the correct value. This type of mathematical reasoning is a known weakness of LLMs [24].

Sometimes the model generates correctly-optimized code but fails to produce the pass list needed to achieve it. Listing 4 shows one such example. A further class of error is when the model makes unsafe optimizations by failing to analyze the input code. Listing 5 shows an example.

We observe an interesting connection between the quality of pass lists and the corresponding optimized code, shown in Figure 6. When the model produces a poor-performing pass list, the quality of the generated code is lower.

Run passes ***-instcombine -simplifycfg*** *to reduce instruction count from* ***14*** *to* ***7:***

```
define dso_local i32 @f1(i32 %0) {
 %2 = load i64, i64* getelementptr inbounds(
   %struct.t2, %struct.t2* @gvar, i64 0, i32 0), align 8
 %3 = icmp eq i64 %2, 0
 %4 = icmp eq i32 %0, 0
 %or.cond = or i1 %3, %4
 %5 = load i32, i32* @S64_MAX, align 4
 %6 = select i1 %or.cond, i32 %5, i32 %0
 ret i32 %6
}
```

Listing 4: An example where the model generates correctly optimized code but fails to produce the pass list needed to produce the desired code. The model-optimized code and instruction count predictions correctly match the performance of the autotuner, but the model omitted the `-mem2reg` pass needed to achieve this code. The model-generated pass list yields 10 instructions instead of 7.

```
define i32 @f1(
  i32 %0,
  i32 %1
) align 2 {
 br label %3

3:
 %i = phi i32 [%7, %6], [2, %2]
 %4 = mul nsw i32 %i, %i
 %5 = icmp sgt i32 %4, %1
 br i1 %5, label %8, label %6

6:
 %7 = add i32 %i, 1
 br label %3

8:
 ret i32 2
}
```

(a) Desired optimized code.

```
int f1(int x, int y) {
  int i = 2;
  while (i * i < y) {
    i += 1;
  }
  return 2;
}
```

(b) Equivalent (hand-written) C code.

```
define i32 @f1(
  i32 %0,
  i32 %1
) align 2 {
 ret i32 2
}
```

(c) Model-optimized code.

Listing 5: An example of an unsafe optimization by the model. The 33-instruction input program (not shown) contains a loop that is not always safe to optimize away. For example, when `y = INT_MAX` the loop never terminates.

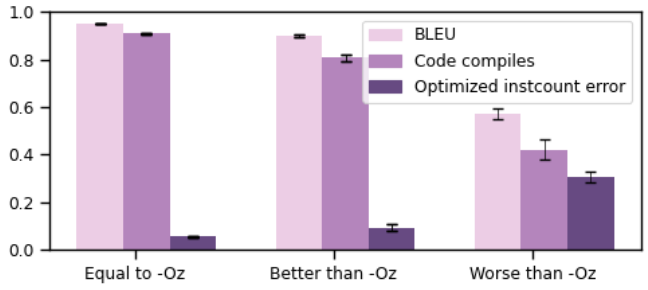


Figure 6: Model-optimized code quality as a function of the performance of the generated pass list. Code quality is lower when the pass list performs worse than -Oz. The model-optimized code resembles the ground truth less (lower BLEU score), the code is less likely to compile, and the model struggles to estimate the instruction count (higher error). Error bars show 95% confidence intervals.

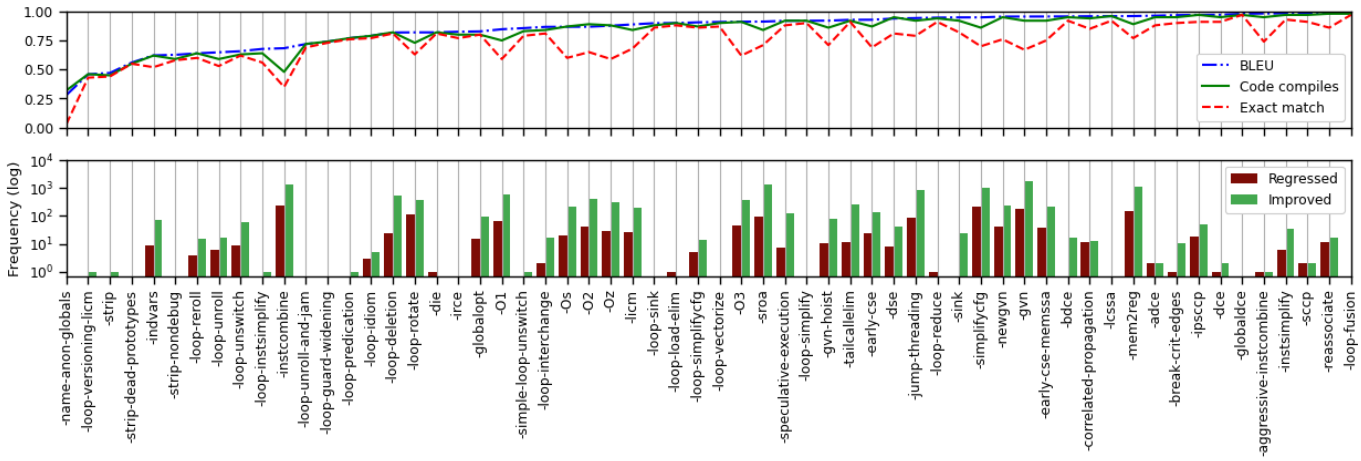


Figure 7: Training a model to predict single optimization passes. The top subplot evaluates the quality the of generated code for the corresponding pass (ordered by BLEU score). The bottom subplot shows the frequency that the corresponding pass contributed to an improvement or regression of instruction count over -Oz.

Figure 8: Ablating the impact of training data size and the auxiliary co-training task of generating optimized code (denoted *No Aux*). Data size is measured as a number of training examples. The graph shows performance on a holdout validation set during training.

Table VI: Ablation experiments. We evaluate the impact of varying training data size and of training the model to generate the optimized code. We train each model for 30k steps and report performance of the best model checkpoint on a holdout validation set of 1,000 unseen IR functions.

n training examples	generate optimized code?	overall improvement
1,000,000	✓	4.95% (—)
500,000	✓	3.91% (-21%)
250,000	✓	3.74% (-24%)
1,000,000	×	4.15% (-16%)

V. Additional Experiments

In the previous section, we evaluated the performance of an LLM trained to optimize LLVM-IR for code size. In this section, we build additional models to better understand the properties of LLMs for code optimization. All models use the same architecture and parameters as in Section III.

A. Abalation of Dataset Size

We ablate the contribution of dataset size by training two additional models and varying the amount of the training data from 50% (500k examples) down to 25% (250k examples) by random dropout. Figure 8 shows progress during the training of the models. For dataset sizes of 50% and 25%, the models begin to overfit the training set after around 8B training tokens. Table VI shows the peak performance of each configuration. With 50% and 25% of the training data, downstream performance falls by 21% and 24%, respectively.

B. Abalation of Code Optimization Task

We train the model to generate not just a pass list but also the optimized code resulting from this pass list. One may expect this to degrade model performance – not only must it learn to predict good pass lists, but also how to produce correctly optimized code, a more difficult task. In fact, we believe this to be crucial to model performance. By forcing LLMs to learn the semantics of LLVM-IR we enable them to make better optimization decisions.

To ablate this we trained a model to generate only pass lists without the corresponding optimized code. We kept the data mix and all other parameters the same. Figure 8 and Table VI show that without training the model to generate optimized code, downstream performance falls by 16%.

C. Evaluation of Single Pass Translation

In previous sections we trained LLMs to orchestrate optimization passes to produce the best-optimized code. In this section, we evaluate the ability of LLMs to emulate the different optimizations in themselves. For this experiment, the model input is an unoptimized IR and the name of an optimization pass to apply, the output is the IR after applying this pass.

Dataset We generate a new dataset for this task using 60 optimization passes and applying them randomly to the programs from Table I. We augment the dataset of unoptimized

```
Optimize the following LLVM-IR using -name-anon-globals:

@0 = private
@anon.2ef3bda806391c61822366a2a59f2569.0 = private
@anon.95277a486ffed0b6ba33ab3385b3d7bd.0 = private
 ↪unnamed_addr constant [14 x i8] c"<snip>", align 1

define dso_local i32 @f1(i8* %0) {
%2 = call i32 @f2(i8* %0, i8* getelementptr inbounds(
 ↪[14 x i8], [14 x i8]*
 ↪@0,
   @anon.2ef3bda806391c61822366a2a59f2569.0,
   @anon.95277a486ffed0b6ba33ab3385b3d7bd.0,
  ↪i64 0, i64 0))
 ret i32 %2
}
```

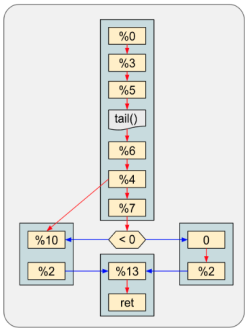
(a) Failure due to incomplete information. The `-name-anon-globals` pass uses the module name to compute a hash. Lacking this, the model hallucinates a random hash.

```
Optimize the following LLVM-IR using -instcombine:

@var_12 = external dso_local global i64, align 8
@var_13 = external dso_local global i32, align 4
@var_14 = external dso_local global i32, align 4

define dso_local void @f1(i64 %arg) {
 %tmp = alloca i64, align 8
 store i64 %arg, i64* %tmp, align 8
 %tmp1 = load i64, i64* %tmp, align 8
 %tmp2 = sub i64 0, %tmp1
 %tmp3 = sub i64 0, %tmp2
 store i64 %tmp3, i64* @var_12, align 8
 store i64 %arg, i64* @var_12, align 8
 store i64 0,    i64* @var_12, align 8
 store i32 1, i32* @var_13, align 4
 store i32 0, i32* @var_14, align 4
 ret void
}
```

(b) Failed data-flow analysis. The model correctly removes redundant instructions but substites the wrong value for a variable. The model-optimized code compiles and has a high BLEU score, but is incorrect.

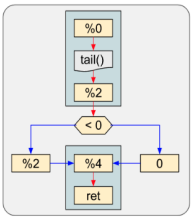
Listing 6: Example failures from the pass translation experiment. We combine the model input (red), ground-truth (blue), and model-generated (green) texts into a single unified diff for brevity. Black text is common to all three.

```
Optimize the following LLVM-IR using -Oz:

%s1 = type { i32 }
@"llvm.used" = appending global [1 x i8*]  [i8*
 bitcast(i32(%s1*)* @f1 to i8*)], section "llvm.metadata"

define dso_local i32 @f1(%s1* %0) {
 %2 = alloca i32, align 4
 %3 = alloca %s1*, align 8
 %4 = alloca i32, align 4
 store %s1* %0, %s1** %3, align 8
 %5 = load %s1*, %s1** %3, align 8
 %6 = tail call i32 @f2(%s1* %5)
 store i32 %6, i32* %4, align 4
 %7 = load i32, i32* %4, align 4
 %8 = icmp slt i32 %7, 0
 br i1 %8, label %9, label %11
9:
 %10 = load i32, i32* %4, align 4
 store i32 %10, i32* %2, align 4
 br label %12
11:
 store i32 0, i32* %2, align 4
 br label %12
12:
 %13 = load i32, i32* %2, align 4
 ret i32 %13
}
```



(a) Model prompt.

```
%s1 = type { i32 }
define dso_local i32 @f1(%s1* %0) {
 %2 = tail call i32 @f2(%s1* %0)
 %3 = icmp slt i32 %2, 0
 %4 = select i1 %3, i32 %2, i32 0
 ret i32 %4
}
```



(b) Model-optimized code.

Listing 7: Example of correct generation of optimized IR. The model performed several complex optimizations including control-flow simplification and replacing if-then-else code blocks with instructions.

code with partially optimized code by first running a sequence of randomly selected passes on unoptimized IRs before the desired target pass. We collect 10,000 unique (prompt, answer) examples for each of the 60 passes for a total of 600k examples.

Model We trained a new model from scratch on this pass translation dataset. It reached peak performance after 11B training tokens (74 GPU days).

Results Figure 7 summarizes model performance. The average BLEU score over all passes is 0.846, with exact character-by-character matches 73.7% of the time and compilable code 82.3% of the time. We also plot the frequency with which each of the optimizations appears in a model-generated pass list that improved or regressed performance over -Oz in Table III. We find no correlation between code quality metrics and its frequency in generated pass lists.

As can be seen, many passes are learned near-perfectly while others prove more challenging. Of the passes that perform poorly, some of them hint at simple improvements to the representation while others result from deeper limitations of the model's reasoning. Listing 6a shows an example from the `-name-anon-globals` pass, which is a simple utility pass that renames anonymous global variables using a hash of the module name. Since we do not provide the module name in the prompt, the LLM is forced to hallucinate random values. We will add the module name to prompts to address this.

Listing 6b shows an example from the `-instcombine` pass. This is a complex pass that is implemented in over 4.5k lines of C++ code in LLVM. We see that the model correctly identifies the instructions to combine, but makes an error in data flow analysis and substitutes an incorrect value. This is an important optimization that frequently occurs in pass lists that outperform -Oz. We will explore an active learning approach in which more examples are provided for complex and difficult passes.

Finally, we present an example of correct model optimization in Listing 7. The example combines several non-trivial code manipulations: register allocation, control flow graph simplification, and instruction combining. We visualize the control- and data-flow graphs to help interpret the changes that the model made. Even on the scale of these small IR functions, we find the sophisticated grasp of LLVM-IR semantics demonstrated by the LLM remarkable. The model has learned to perform these optimizations entirely from examples, without access to the compiler implementation.

VI. Discussion

We have shown that LLMs can near-perfectly emulate many compiler optimizations and outperform prior approaches, but there are limitations. This section aims to provide a pragmatic discussion of limits and directions for future research.

A. Context Window

The main limitation of LLMs is the limited sequence length of inputs (context window). In this work we target 2k-token context windows and split IRs into individual functions to maximize the amount of code we can fit into the context window. This is undesirable for a number of reasons. First, it limits the context available to the model when making optimization decisions; second, it prevents intra-function optimization; third, we cannot optimize code that does not fit within the context window, and Figure 5 suggests that larger programs have more interesting optimization opportunities.

Researchers are adopting ever-increasing context windows [45], but finite context windows remain a common concern with LLMs. As new techniques for handling long sequences continue to evolve we plan to incorporate them and apply them to code optimization, e.g. Code Llama's variant of positional interpolation [46] which is RoPE base period scaling [9] or recent length extrapolation techniques [47].

B. Math Reasoning and Logic

Compilers perform lots of arithmetic. Whenever possible expressions are evaluated at compile time to minimize work at runtime and to enable better optimization decisions. We see examples of LLMs struggling with this type of reasoning, e.g. failed constant folding (Listing 3) and failed data-flow analysis (Listing 6b).

We think that a chain-of-thought approach [48] in which models are trained to break complex reasoning problems down into incremental steps will prove fruitful. We took the first step in this direction by breaking optimizations down into individual passes in Section V-C. We also plan to focus training on a curriculum of arithmetic and logic, and train LLMs that use tools to compute intermediate results [49, 50].

C. Inference Speed

Compilers are fast. It takes two orders of magnitude more time for the model to generate a pass list than it does for the compiler to execute it. While this is much faster than the autotuner it is trained on, it remains an overhead that may prove prohibitive for some applications. That is to say nothing of the difference in compute resources needed to evaluate compiler heuristics vs. a 7B-parameter LLM on multiple GPUs.

In addition to aggressive batching and quantization [51], significant inference speedups can be achieved by specializing the vocabulary to a use case. For example, we can reduce entire subsequences of passes to single vocabulary elements using Byte Pair Encoding so that at inference time fewer tokens need to be generated.

VII. Related Work

Compiler pass ordering for performance has been exploited for decades [26, 52, 53]. Over the years there have been many approaches using machine learning [18–20, 39, 54, 55]. The application of machine learning in compilers is not limited to pass order and has been applied to many other problems [17, 56–59]. No one has applied LLMs to the problem of pass ordering, we are the first to do so.

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 [11], assembly to C [36, 60], and source-to-source transpilation [10]. In these works code correctness cannot be guaranteed. In our work we use code generation solely as an auxiliary learning task – correctness is supplied by the compiler.

Language models have found broad adoption for coding tasks, though few operate at the level of compiler IR. Gallagher et al. train a RoBERTA architecture on LLVM-IR for the purpose of code weakness identification [61] and Transcoder-IR [12] uses LLVM-IR as a pivot point for source-to-source translation. Neither use LLMs to optimize the IR as we do.

Many language models have been trained on source code including CodeBERT [62], GraphCodeBERT [63], and CodeT5 [64] which are trained to perform multiple tasks including code search, code summarization, and documentation generation. LLMs trained on source code have also been used for program fuzzing [13, 14, 65], test generation [15], and automated program repair [66–68]. A large number of useful applications have been explored for language models, however, this is the first work where an LLM is used specifically for optimizing code.

Most LLMs are trained at least partly on code [3, 5, 25, 69]. Some LLMs are trained similarly to general models but especially target programming languages and can be used for code completion such as Codex [8] which powers Copilot [70]. The introduction of fill-in-the-middle capabilities is especially useful for real-world code completion use cases and has become common in recent code models such as InCoder [6], SantaCoder [4], StarCoder [1], and Code Llama [9]. Code Llama was also trained to follow instructions and generate code as well as explain its functionalities.

While the multi-terabyte training corpora for these models contain some assembly, we believe that a focused exploration of the value of LLMs in the domain of compilers will be of value to the community. This paper aims to provide that.

VIII. Conclusions

We present the first steps towards LLMs for code optimization. We construct a model that can predict good optimization strategies for unseen LLVM-IR. Results are promising, though we face challenges in sequence length which limits us to operating over small program fragments, and in arithmetic reasoning which limits the ability of the model to predict the outcome of optimizations. We hope to inspire the research community to push beyond LLMs for simple max-likelihood code generation and into performance-aware code optimization.

REFERENCES

[1] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim, Q. Liu, E. Zheltonozhskii, et al. "StarCoder: may the source be with you!" In: *arXiv:2305.06161* (2023).

[2] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. D. Lago, T. Hubert, P. Choy, et al. "Competition-Level Code Generation with AlphaCode". In: *Science* 378.6624 (2022).

[3] OpenAI. "GPT-4 Technical Report". In: *arXiv:2303.08774* (2023).

[4] L. B. Allal, R. Li, D. Kocetkov, C. Mou, C. Akiki, C. M. Ferrandis, N. Muennighoff, M. Mishra, A. Gu, M. Dey, L. K. Umapathi, C. J. Anderson, et al. "SantaCoder: don't reach for the stars!" In: *arXiv:2301.03988* (2023).

[5] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann, P. Schuh, K. Shi, et al. "PaLM: Scaling Language Modeling with Pathways". In: *arXiv:2204.02311* (2022).

[6] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, W.-t. Yih, L. Zettlemoyer, and M. Lewis. "InCoder: A Generative Model for Code Infilling and Synthesis". In: *arXiv:2204.05999* (2023).

[7] S. Gunasekar, Y. Zhang, J. Aneja, C. C. T. Mendes, A. D. Giorno, S. Gopi, M. Javaheripi, P. Kauffmann, G. de Rosa, O. Saarikivi, A. Salim, S. Shah, et al. "Textbooks Are All You Need". In: *arXiv:2306.11644* (2023).

[8] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, et al. "Evaluating Large Language Models Trained on Code". In: *arXiv:2107.03374* (2021).

[9] B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin, A. Kozhevnikov, I. Evtimov, et al. "Code Llama: Open Foundation Models for Code". In: *arXiv:2308.12950* (2023).

[10] M.-A. Lachaux, B. Roziere, L. Chanussot, and G. Lample. "Unsupervised Translation of Programming Languages". In: *arXiv:2006.03511* (2020).

[11] J. Armengol-Estapé and M. F. O'Boyle. "Learning C to x86 Translation: An Experiment in Neural Compilation". In: *arXiv:2108.07639* (2021).

[12] M. Szafraniec, B. Roziere, F. Charton, H. Leather, P. Labatut, and G. Synnaeve. "Code Translation with Compiler Representations". In: *arXiv:2207.03578* (2022).

[13] G. Ye, Z. Tang, S. H. Tan, S. Huang, D. Fang, X. Sun, L. Bian, H. Wang, and Z. Wang. "Automated conformance testing for JavaScript engines via deep compiler fuzzing". In: *PLDI*. 2021.

[14] Y. Deng, C. S. Xia, H. Peng, C. Yang, and L. Zhang. "Large Language Models Are Zero-Shot Fuzzers: Fuzzing Deep-Learning Libraries via Large Language Models". In: *ISSTA*. 2023.

[15] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip. "Adaptive Test Generation Using a Large Language Model". In: *arXiv:2302.06527* (2023).

[16] OpenAI. *ChatGPT*. https://chat.openai.com/.

[17] M. Trofin, Y. Qian, E. Brevdo, Z. Lin, K. Choromanski, and D. Li. "MLGO: a Machine Learning Guided Compiler Optimizations Framework". In: *arXiv:2101.04808* (2021).

[18] Z. Wang and M. O'Boyle. "Machine Learning in Compiler Optimisation". In: *arXiv:1805.03441* (2018).

[19] H. Leather and C. Cummins. "Machine Learning in Compilers: Past, Present and Future". In: *FDL*. 2020.

[20] Y. Liang, K. Stone, A. Shameli, C. Cummins, M. Elhoushi, J. Guo, B. Steiner, X. Yang, P. Xie, H. Leather, and Y. Tian. "Learning Compiler Pass Orders using Coreset and Normalized Value Prediction". In: *ICML*. 2023.

[21] C. Cummins, Z. Fisches, T. Ben-Nun, T. Hoefler, M. O'Boyle, and H. Leather. "ProGraML: A Graph-based Program Representation for Data Flow Analysis and Compiler Optimizations". In: *ICML*. 2021.

[22] C. Lattner and V. Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: *CGO*. 2004.

[23] N. Asher, S. Bhar, A. Chaturvedi, J. Hunter, and S. Paul. "Limits for Learning with Language Models". In: *arXiv:2306.12213* (2023).

[24] J. Qian, H. Wang, Z. Li, S. Li, and X. Yan. "Limitations of Language Models in Arithmetic and Symbolic Induction". In: *arXiv:2208.05051* (2022).

[25] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, et al. "Llama 2: Open Foundation and Fine-Tuned Chat Models". In: *arXiv:2307.09288* (2023).

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

[27] C. Cummins, B. Wasti, J. Guo, B. Cui, J. Ansel, S. Gomez, S. Jain, J. Liu, O. Teytaud, B. Steiner, Y. Tian, and H. Leather. "CompilerGym: Robust, Performant Compiler Optimization Environments for AI Research". In: *CGO*. 2022.

[28] D. Kocetkov, R. Li, L. B. Allal, J. Li, C. Mou, C. M. Ferrandis, Y. Jernite, M. Mitchell, S. Hughes, T. Wolf, et al. "The Stack: 3TB of Permissively Licensed Source Code". In: *arXiv:2211.15533* (2022).

[29] L. Gao, S. Biderman, S. Black, L. Golding, T. Hoppe, C. Foster, J. Phang, H. He, A. Thite, N. Nabeshima, et al. "The Pile: An 800GB Dataset of Diverse Text for Language Modeling". In: *arXiv:2101.00027* (2020).

[30] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt. "CodeSearchNet Challenge: Evaluating the State of Semantic Code Search". In: *arXiv:1909.09436* (2019).

[31] A. Fadel, H. Musleh, I. Tuffaha, M. Al-Ayyoub, Y. Jararweh, E. Benkhelifa, and P. Rosso. "Overview of the PAN@FIRE 2020 task on the authorship identification of SOurce COde (AI-SOCO)". In: *FIRE*. 2020.

[32] J. Armengol-Estapé, J. Woodruff, A. Brauckmann, J. W. d. S. Magalhães, and M. O'Boyle. "ExeBench: an ML-scale Dataset of Executable C Functions". In: *MAPS*. 2022.

[33] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin. "Convolutional Neural Networks Over Tree Structures for Programming Language Processing". In: *AAAI*. 2016.

[34] X. Yang, Y. Chen, E. Eide, and J. Regehr. "Finding and Understanding Bugs in C Compilers". In: *PLDI*. 2011.

[35] V. Livinskii, D. Babokin, and J. Regehr. "Random Testing for C and C++ Compilers with YARPGen". In: *OOPSLA*. 2020.

[36] J. Armengol-Estapé, J. Woodruff, C. Cummins, and M. F. O'Boyle. "SLaDe: A Portable Small Language Model Decompiler for Optimized Assembler". In: *arXiv:2305.12520* (2023).

[37] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. "Attention Is All You Need". In: *NeurIPS* (2017).

[38] P. Gage. "A New Algorithm for Data Compression". In: *C Users Journal* 12.2 (1994).

[39] A. Haj-Ali, Q. Huang, W. Moses, J. Xiang, J. Wawrzynek, K. Asanovic, and I. Stoica. "AutoPhase: Juggling HLS Phase Orderings in Random Forests with Deep Reinforcement Learning". In: *MLSys*. 2020.

[40] I. Loshchilov and F. Hutter. "Decoupled Weight Decay Regularization". In: *arXiv:1711.05101* (2017).

[41] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu. "BLEU: A Method for Automatic Evaluation of Machine Translation". In: *ACL*. 2002.

[42] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. "Proximal Policy Optimization Algorithms". In: *arXiv:1707.06347* (2017).

[43] M. Paszkowski. *LLVM Canon*. https : / / github . com / michalpaszkowski/LLVM-Canon.

[44] W. M. McKeeman. "Differential Testing for Software". In: *Digital Technical Journal* 10.1 (1998).

[45] J. Ding, S. Ma, L. Dong, X. Zhang, S. Huang, W. Wang, and F. Wei. "LongNet: Scaling Transformers to 1,000,000,000 Tokens". In: *arXiv:2307.02486* (2023).

[46] S. Chen, S. Wong, L. Chen, and Y. Tian. "Extending Context Window of Large Language Models via Positional Interpolation". In: *arXiv:2306.15595* (2023).

[47] Y. Sun, L. Dong, B. Patra, S. Ma, S. Huang, A. Benhaim, V. Chaudhary, X. Song, and F. Wei. "A Length-Extrapolatable Transformer". In: *arXiv:2212.10554* (2022).

[48] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou, et al. "Chain-of-thought prompting elicits reasoning in large language models". In: *NeurIPS*. 2022.

[49] L. Gao, A. Madaan, S. Zhou, U. Alon, P. Liu, Y. Yang, J. Callan, and G. Neubig. "Pal: Program-aided language models". In: *ICML*. 2023.

[50] K. Cobbe, V. Kosaraju, M. Bavarian, M. Chen, H. Jun, L. Kaiser, M. Plappert, J. Tworek, J. Hilton, R. Nakano, et al. "Training Verifiers to Solve Math Word Problems". In: *arXiv:2110.14168* (2021).

[51] G. Xiao, J. Lin, M. Seznec, H. Wu, J. Demouth, and S. Han. "SmoothQuant: Accurate and Efficient Post-Training Quantization for Large Language Models". In: *ICML*. 2023.

[52] F. Bodin, T. Kisuki, P. Knijnenburg, M. O'Boyle, and E. Rohou. "Iterative Compilation in a Non-linear Optimisation Space". In: *FDO*. 1998.

[53] T. Kisuki, P. Knijnenburg, and M. O'Boyle. "Combined Selection of Tile Sizes and Unroll Factors using Iterative Compilation". In: *PACT*. 2000.

[54] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. O'Boyle, J. Thomson, M. Toussaint, and C. Williams. "Using Machine Learning to Focus Iterative Optimization". In: *CGO*. 2006.

[55] W. F. Ogilvie, P. Petoumenos, Z. Wang, and H. Leather. "Minimizing the Cost of Iterative Compilation with Active Learning". In: *CGO*. 2017.

[56] A. H. Ashouri, M. Elhoushi, Y. Hua, X. Wang, M. A. Manzoor, B. Chan, and Y. Gao. "MLGOPerf: An ML Guided Inliner to Optimize Performance". In: *arXiv:2207.08389* (2022).

[57] A. Haj-Ali, N. K. Ahmed, T. Willke, S. Shao, K. Asanovic, and I. Stoica. "NeuroVectorizer: End-to-End Vectorization with Deep Reinforcement Learning". In: *CGO*. 2020.

[58] C. Cummins, P. Petoumenos, Z. Wang, and H. Leather. "End-to-End Deep Learning of Optimization Heuristics". In: *PACT*. 2017.

[59] P. M. Phothilimthana, A. Sabne, N. Sarda, K. S. Murthy, Y. Zhou, C. Angermueller, M. Burrows, S. Roy, K. Mandke, R. Farahani, et al. "A Flexible Approach to Autotuning Multi-pass Machine Learning Compilers". In: *PACT*. 2021.

[60] I. Hosseini and B. Dolan-Gavitt. "Beyond the C: Retargetable Decompilation using Neural Machine Translation". In: *arXiv:2212.08950* (2022).

[61] S. K. Gallagher, W. E. Klieber, and D. Svoboda. *LLVM Intermediate Representation for Code Weakness Identification*. 2022.

[62] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, et al. "CodeBERT: A Pre-trained Model for Programming and Natural Languages". In: *arXiv:2002.08155* (2020).

[63] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou. "GraphCodeBERT: Pre-training Code Representations with Data Flow". In: *arXiv:2009.08366* (2021).

[64] Y. Wang, W. Wang, S. Joty, and S. C. Hoi. "CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation". In: *arXiv:2109.00859* (2021).

[65] C. S. Xia, M. Paltenghi, J. L. Tian, M. Pradel, and L. Zhang. "Universal Fuzzing via Large Language Models". In: *arXiv:2308.04748* (2023).

[66] C. S. Xia and L. Zhang. "Less Training, More Repairing Please: Revisiting Automated Program Repair via Zero-shot Learning". In: *arXiv:2207.08281* (2022).

[67] C. S. Xia, Y. Wei, and L. Zhang. "Automated Program Repair in the Era of Large Pre-Trained Language Models". In: *ICSE*. 2023.

[68] C. S. Xia and L. Zhang. "Keep the Conversation Going: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT". In: *arXiv:2304.00385* (2023).

[69] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, et al. "Llama: Open and efficient foundation language models". In: *arXiv preprint arXiv:2302.13971* (2023).

[70] GitHub. *Copilot*. https://copilot.github.com/.