LEARNING PERFORMANCE-IMPROVING CODE EDITS 🥧

Aman Madaan[1], **Alexander Shypula**[2], **Uri Alon**[1], **Milad Hashemi**[3],
Parthasarathy Ranganathan[3], **Yiming Yang**[1], **Graham Neubig**[1,4], **Amir Yazdanbakhsh**[5]
[1]Language Technologies Institute, Carnegie Mellon University [2]University of Pennsylvania
[3]Google [4]Inspired Cognition [5]Google Research, Brain Team
amadaan@cs.cmu.edu, shypula@seas.upenn.edu, ualon@cs.cmu.edu
miladh@google.com, parthas@google.com, yiming@cs.cmu.edu
gneubig@cs.cmu.edu, ayazdan@google.com

ABSTRACT

The waning of Moore's Law has shifted the focus of the tech industry towards alternative methods for continued performance gains. While optimizing compilers are a standard tool to help increase program efficiency, programmers continue to shoulder much responsibility in crafting and refactoring code with better performance characteristics. In this paper, we investigate the ability of large language models (LLMs) to suggest *functionally correct, performance improving* code edits. We hypothesize that language models can suggest such edits in ways that would be impractical for static analysis alone. We investigate these questions by curating a large-scale dataset of **P**erformance-**I**mproving **E**dits, PIE. PIE contains trajectories of programs, where a programmer begins with an initial, slower version and iteratively makes changes to improve the program's performance. We use PIE to evaluate and improve the capacity of large language models. Specifically, we use examples from PIE to fine-tune multiple variants of CODEGEN, a billion-scale Transformer-decoder model. Additionally, we use examples from PIE to prompt OpenAI's CODEX using a few-shot prompting. By leveraging PIE, we find that both CODEX and CODEGEN can generate performance-improving edits, with speedups of more than 2.5× for over 25% of the programs, for C++ and Python, even after the C++ programs were compiled using the O3 optimization level. Crucially, we show that PIE allows CODEGEN, an open-sourced and 10× smaller model than CODEX, to match the performance of CODEX on this challenging task. Overall, this work opens new doors for creating systems and methods that can help programmers write efficient code.[1]

1 INTRODUCTION

Ensuring efficiency is a crucial aspect of programming, particularly when computational resources are limited or the program is utilized at a large scale. The traditional tool for improving program efficiency is the optimizing compiler, which iteratively applies optimizations as an intermediate step in generating machine code (Aho et al., 2007). Despite the impressive progress in optimizing compilers, programmers are still generally responsible for numerous performance considerations. Experienced programmers can often shave off microseconds from an already optimized code. However, such improvements typically come after laborious consideration of factors such as algorithm selection, data structure choice, memory hierarchy, and expected runtime inputs.

Large language models (LLMs) have shown great potential in a variety of software-related tasks ranging from competitive programming (Li et al., 2022), code completion/editing (Fried et al., 2022), to clone detection, defect detection, and much more (Lu et al., 2021; Xu et al., 2022; Zhou et al., 2022; Austin et al., 2021).

[1]Code and dataset are available at https://pie4perf.com/.

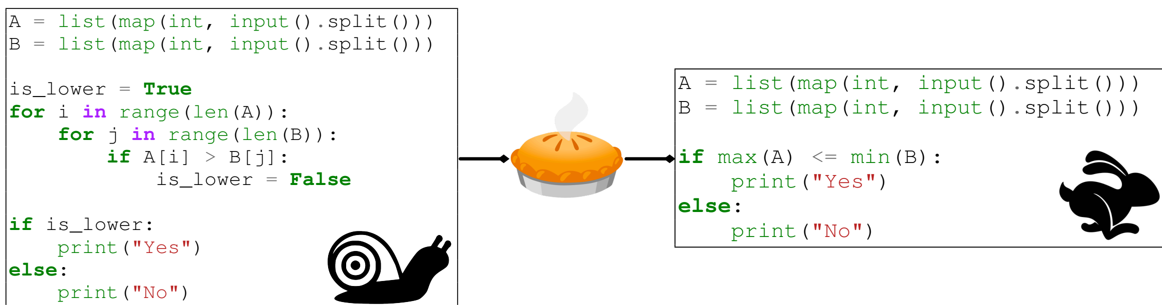

Figure 1: An example of a program optimized with CODEX. The slow version (left) uses two nested loops to compare each element in A to each element in B, which takes $\mathcal{O}(N^2)$ time. The faster version (right) is using two built-in functions (max and min) to compare the maximum value in A to the minimum value in B, which takes $\mathcal{O}(N)$ time. This example is illustrative and does *not* exist in the PIE dataset.

Motivated by these successes, we seek to ask and answer the following question: can LLMs *improve the efficiency of programs* while operating at the level of high-level programming languages like Python or C++?

We hypothesize that large language models have the capacity to propose rewrites that would be *impractical or very non-trivial to expect from an optimizing compiler or search procedure*. If so, large language models may have an important role to play in assisting programmers in choosing more desirable refactorings.

We set forth to evaluate and improve the capacity of large language models to improve programs by curating a dataset of **P**erformance-**I**mproving **E**dits, PIE. We collect trajectories of programs written by the same user, where we track a single programmer's submission, how it evolved over time, and its performance characteristics. Having programs that were written by the same user for the same problem is important because it provides us with pairs of programs that are mostly identical, except for the few, and targeted differences that have an outsize impact on the program's runtime.

We show that PIE allows us to improve the efficacy of large language models for code optimization. Specifically, we investigate the effects of using PIE for few-shot prompting on CODEX and fine-tuning models like CODEGEN. We see noticeable improvements in all experiments using PIE. Ultimately, these models can successfully perform substantial (up to $2.5\times$) code optimization for Python and C++ for many examples (up to 50% of the test set). We also demonstrate that CODEGEN-2B and CODEGEN-16B trained on PIE can, in some cases, match the performance of CODEX while being up to $10\times$ smaller in size than CODEX.

2 A DATASET OF PERFORMANCE-IMPROVING CODE EDITS

We introduce PIE, a performance-oriented dataset across multiple programming languages. Importantly, PIE captures the progressive changes made by a programmer over time to improve their code. Additionally, all programs in our dataset come with unit tests that can be used to both access functional accuracy and as well as real, measured runtime. We create our dataset by branching from the CodeNet (Puri et al., 2021) collection of code samples.[2]

Our dataset is based on the insight that iterative edits made by the same developer are likely to reflect improvement while retaining the developer's stylistic choices (e.g., identifier names and program structure). In the context of few-shot prompting, we expect this format will provide clear instructions to a language model. In

[2]https://github.com/IBM/Project_CodeNet

the context of fine-tuning, we hypothesize that the learning task would be easier than using program pairs from different developers and would preserve many stylistic elements.

We differentiate our work from others' across two separate dimensions. First, our dataset and our experiments are unique in that we execute our programs to determine functional accuracy **as well as runtime**. To our knowledge, ours is the first work involving high-level programming languages to rely on such a degree of benchmarking. Other works either rely on manual human review (Garg et al., 2022), which is hard to scale or use proxies of latency for low-level (Shypula et al., 2021) or domain-specific (Shi & Zhang, 2020) programming languages. Secondly, our dataset is novel because we create (slow, fast) program pairs written by the same-exact programmer. Recent work (Chen et al., 2022b) has attempted to construct such a dataset by taking pairs from different programmers and canonicalizing all identifiers. In addition to the challenges posed to evaluation, canonicalization can dramatically affect program semantics, to the point where programs may not compile or execute without additional post-processing.

2.1 Dataset Construction

Programmers typically write multiple versions of programs for a given problem statement. Let $\mathbb{Y} = \{y_1, ..., y_n\}$ be chronologically sorted programs, written by the user u for a problem statement x (e.g., *calculate the sum of n numbers*). These versions can differ in terms of correctness or time efficiency (e.g., y_{i+1} may pass more unit tests than y_i, or could be faster), but typically subsequent versions tend to improve performance characteristics.

Crucially, the differences in subsequent versions are typically minor — programmers usually change only small portions that affect correctness or efficiency, leaving the overall structure intact. From the set of submissions $\mathbb{Y}$, we remove programs that were not "*Accepted*" by the automated system; this eliminates all programs that are either incorrect or take more than the allocated time to run. We then sort the set of remaining programs chronologically, yielding a *trajectory* of programs $\mathbb{Y}^* = [y_1^*, y_2^*, \ldots, y_n^*]$. Each program $y_i^* \in \mathbb{Y}^*$ is "*Correct*" in the sense that it passes all unit tests and runs within the allocated time limit. While there are multiple ways of harnessing the trajectory for a variety of problems, our focus in this work is learning to *optimize* programs. Thus, we extract (slower, faster) pairs of programs from the given trajectories.

To this end, we first divide a trajectory into pairs of the form (y_i, y_{i+1}); that is: $\mathbb{P} = (y_1, y_2), (y_2, y_3), \ldots, (y_{n-1}, y_n)$. Then, we apply a series of filtering steps on $\mathbb{P}$. Specifically, we keep pairs where the number of non-empty lines is lower than 150 and the number of different lines is less than 50%. Next, we keep pairs (y_i, y_{i+1}) for which the relative time improvement is more than 10%, that is: $\frac{(\texttt{time}(y_i) - \texttt{time}(y_{i+1}))}{\texttt{time}(y_i)} > 10\%$ where $\texttt{time}\,(y)$ denotes the runtime of the program y. We then apply some language-specific filters to Python, such as removing unused libraries that may affect the runtime unexpectedly. Specifically, we found that certain pairs (y_i, y_{i+1}) for Python show speedup because of uninteresting reasons, such as unused imports in y_i that are removed in y_{i+1}. We use Autoflake[3] to detect and remove such instances. Finally, we compile the filtered $\mathbb{P}$ for all unique pairs of (u, x) in our dataset, and split them to train/test/validation subsets, while ensuring that solutions for any problem only appear in one of the splits. Table 1 shows statistics of the resulting dataset, and we provide additional metadata in Appendix H.

3 Learning to Improve Code Performance

We perform experiments utilizing both *compiled* and *interpreted* programming languages: C++ and Python. In a compiled language like C++, we are also interested in demonstrating improvements in runtime on top of the compiler's optimization passes (e.g., by using the –O3 optimization flag). In addition to these two languages, we also experiment with two broad classes of improving our models: few-shot prompting and fine-tuning.

[3]https://github.com/PyCQA/autoflake

Language	Train	Val	Test	Rel. Imp.	y_i LOC	y_{i+1} LOC	DIFF LOC
Python	35k	2.5k	1.5k	37.1 ± 26.2	23.4 ± 18.8	22.0 ± 18.4	10.0 ± 8.5
C++	88k	5k	3.7k	48.6 ± 27.0	52.4 ± 30.8	52.9 ± 33.9	20.5 ± 17.2
Java	3.6k	1k	0.2k	30.8 ± 20.3	52.3 ± 35.2	54.8 ± 42.9	15.1 ± 15.3

Table 1: PIE contains pairs of (slower, faster) programs written by the same programmer. The Rel. Imp. column indicates the average relative improvement of the fast code y_{i+1} over its slower counterpart y_i, and $\pm$ are followed by the standard deviation.

Few-Shot Prompting. We use CODEX (`code-davinci-002`) from OpenAI for the few-shot experiments. CODEX is based on the 175B parameter model, GPT-3.[4]. For all few-shot experiments, we create a prompt of the format "slow$_1$ $\rightarrow$ fast$_1$; slow$_2$ $\rightarrow$ fast$_2$, ...". A slow test program is appended to this prompt during inference and supplied to the few-shot model. The prompts are shown in Appendix E.

Fine-Tuning. We employ the CODEGEN (Nijkamp et al., 2022) series of varying sizes for our fine-tuning experiments. While large language models (LLMs) such as CODEX (Chen et al., 2021) and tools like ChatGPT can easily perform a wide range of tasks, they are not open-sourced. As a result, its accessibility and deeper study requiring access to parameters (e.g., for fine-tuning) may be limited. We can investigate the impact of directly fine-tuning CODEGEN on the examples in PIE. CODEGEN is available for both Python (mono) and C++ (multi).

Sampling Code generation tasks often benefit from generating many samples for each input (Li et al., 2022). We explore two different forms of sampling strategies: greedy sampling and random sampling. Greedy sampling generates programs by sampling the most likely token at each step. Random sampling sets the softmax temperature to a high value (e.g., 0.7) to generate multiple programs. Every input program in PIE comes with a set of unit tests, allowing us to evaluate each candidate sample for correctness, and select the fastest program among the generated candidates.

4 EVALUATION

We evaluate the capability of LLMs to improve program runtime in two settings, (1) zero-shot and (2) fine-tuning. For the zero-shot setting, we use Codex (Chen et al., 2021) and multiple variants of CodeGen (Nijkamp et al., 2022), from 2B to 16B parameters. Appendix E showcases examples of prompts we used in our evaluations.

4.1 EXPERIMENTAL SETUP

Metrics. In order to evaluate performance improvement, we measure the following metrics for programs that are *functionally correct* and whose *improvements are statistically significant*:

1. **Percent Optimized** $\ulcorner$%OPT$\lrcorner$: The proportion of programs in the test set (out of 1000 unseen samples) that are improved by a certain method.
2. **Speedup** $\ulcorner$SP$\lrcorner$: the absolute improvement in runtime. If o and n are the "old" and "new" runtimes, then SPEEDUP(O, N) $= \left(\frac{o}{n}\right)$.

[4] https://platform.openai.com/docs/model-index-for-researchers

Method	Python			C++ (O3)		
	%OPT	SP (×)	RTR	%OPT	SP (×)	RTR
CODEX (0-shot, greedy)	14.6	1.69	25.78	2.1	2.32	22.86
CODEGEN-16B (1-shot, greedy)	2.2	1.55	25.05	0.2	1.13	11.66
SCALENE	-	-	-	-	-	-
REF	-	-	-	-	-	-
CODEX (greedy)	14.3	**2.7**	**53.49**	2.0	**2.26**	**24.85**
CODEGEN-2B (greedy)	8.2	2.32	48.23	0.2	1.13	11.66
CODEGEN-16B (greedy)	7.9	2.46	51.25	0.9	1.17	14.63
CODEX (16 samples)	38.9	2.61	50.31	8.5	1.69	20.08
CODEGEN-2B (16 samples)	23.2	2.36	45.54	5.5	1.21	15.24
CODEGEN-16B (16 samples)	25.6	2.53	47.6	4.3	1.28	15.89
CODEX (32 samples)	**45.3**	2.62	49.38	**13.0**	1.73	22.55
CODEGEN-2B (32 samples)	27.7	2.34	45.02	8.5	1.24	16.16
CODEGEN-16B (32 samples)	30.3	2.45	46.8	4.7	15.26	1.26

Table 2: Main Results → We compare CODEX, CODEGEN-2B, and CODEGEN-16B in both Python and C++ with greedy decoding and random sampling. We use CODEGEN-mono for Python , and CODEGEN-multi for C++ . For generating multiple samples, we use $\mathcal{T} = 0.7$. The first block shows results from non-finetuned CODEGEN-16B models.

3. **Runtime Reduction** ⌜RTR⌟: the relative, or normalized, improvement in runtime, among the programs which were improved and are functionally correct. Using the same terminology above for runtimes, $\text{RTR}(\text{O}, \text{N}) = \left(\frac{o-n}{o}\right) * 100$. This metric can be thought of "the proportion of runtime has been 'optimized' away." We report the average RTR over the test set.

Functional Correctness. The evaluation of generated programs is a crucial step in our experiment. Our objective is not only to generate optimized programs, but also to generate a program that is functionally and syntactically correct. To evaluate the correctness of the generated program, we run a set of unit tests (5-10 tests). If a single test case fails, we reject the generated program.

Statistical Significance. Variance is an important factor to consider when benchmarking. To account for the variance, we determine if the runtime of optimized programs is statistically significant under Welch's Unequal Variance t-test. [5] In the results, we report the metrics for program pairs whose runtime improvements are statically significant (p-value < 0.05). For all the results, programs that pass both *correctness* and the *significance test* criteria are considered to be successfully optimized in our experiments.

4.2 RESULTS

Table 2 shows our main results: *large language models can optimize the runtime of programs in Python and C++, even after the C++ files were compiled with the O3 optimization level* of gcc. CODEX (with 32 sampled outputs) can improve the runtime of 45.3% of the Python and 13.0% of the C++ examples. CODEGEN

[5] A one-tailed Welch's t-test examines the null hypothesis that the runtime of the model-generated programs have a lower mean than the input program, without the assumption that both populations have equal variance. In order to perform the t-test for every element in our test set, we execute both the input program and the model-generated program 25 times each. We then use the means and standard deviations of the runtime to perform the significance test.

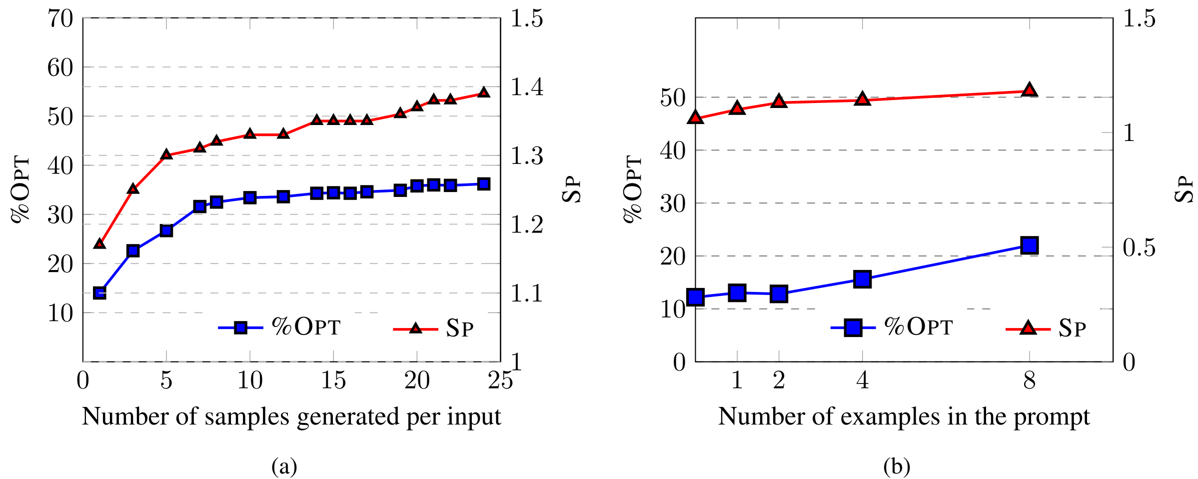

Figure 2: Few-shot prompting has two key design choices: the number of samples that can be drawn for each input, and the number of few-shot examples in the prompt. On the left, we find that drawing a larger number of samples (with fixed examples) consistently helps the coverage ⌜%OPT⌟ and speedup ⌜SP⌟. Figure on the right shows that as more examples are added to the prompt, the model is able to optimize a larger number of programs (increasing ⌜%OPT⌟), with a relatively slow increase in ⌜SP⌟.

models trained on PIE can closely match the performance of CODEX on speedup (⌜SP⌟) and relative runtime reduction ⌜RTR⌟.

Both CODEX and CODEGEN benefit immensely from drawing multiple samples, especially in increasing the number of programs optimized (%OPT increases by $3\times$ with 16 samples). CODEX with greedy decoding has the highest ⌜%OPT⌟ and ⌜RTR⌟ in Python and C++ , with a speedup of $2.7\times$ and $2.26\times$, respectively, and the gains dwindle with a larger number of samples. In contrast, CODEGEN models can increase the %OPT while maintaining or improving the speedup (e.g., CODEGEN-16B for greedy vs. 16 samples). Finally, we observed that validation loss for the CODEGEN-16B-multi model for C++ was still going down at the end of the training, indicating that there may be further room for improvement to improve.

To better understand the capability of CODEX to generate performance-improving code edits, we investigate the impact of the following hyperparameters on the evaluation metrics in Python . We analyze the following configurations: (a) the number of samples drawn from CODEX for each problem in the test set; (b) the number of slow and fast program pairs we include in the prompt to CODEX; and (c) finally, the *type* of prompt.

What is the impact of changing the number of samples drawn from the model? As shown in Figure 2a, both ⌜%OPT⌟ and ⌜SP⌟ monotonically increase as more samples are drawn. The reason is that when additional samples are drawn from the model, the likelihood that at least one of the generated code edits would improve efficiency also proportionally increases.

How effective is PIE when we use fewer examples in the prompt? As shown in Figure 2b, both ⌜%OPT⌟ and ⌜RTR⌟ monotonically increase with more examples (demonstrations) in the prompt. We conclude that the number of examples should be as high as the underlying LM allows, according to its context window size.

Does the magnitude of relative improvement in the prompt examples impact the final result? Finally, Figure 9 (Appendix D) shows that while CODEX is relatively indifferent to the degree of optimization in the

```cpp
int s;
cin >> s;
int a[2000000] = {};
bool flag[10000] = {};
a[0] = s;
flag[a[0]] = true;
int i = 1;
while(1) {
  if (a[i - 1] % 2 == 0) a[i] = a[i - 1] / 2;
  else a[i] = a[i - 1] * 3 + 1;
    if (flag[a[i]]) {
      cout << i + 1 << endl;
      return 0;
    }
  flag[a[i]] = true;
  i++;
}
```

(a) Slower Code.

```cpp
int s;
cin >> s;
map<int, int> seen;
int cnt = 0;
while (seen.find(s) == seen.end()) {
  seen[s] = cnt;
  if (s % 2 == 0) s /= 2;
  else s = s * 3 + 1;
  cnt++;
}
cout << cnt + 1 << endl;
```

(b) Faster Code.

Figure 3: Example of C++ optimizations for a program computing the Collatz sequence length of a particular starting integer generated through prompting CODEX.

prompts, examples at either extreme (90-100% or 1-10% ⌈RTR⌉) are possibly too specific, and hurt performance.

5 ANALYSIS OF GENERATED CODE EDITS

To better understand the nature of generated code edits, we performed a qualitative analysis on the type of suggested optimizations from the evaluated language models. We find that among the numerous statistically significant results, the types of suggested optimizations varied. The optimization types ranges from global refactoring to govern language- and problem-specific optimizations, global refactoring to apply algorithmic changes, to local changes involving memory access and more.

Figure 3 demonstrates an example of a model-generated C++ code edit that moves some computations related to the Collatz sequence[6] from happening on the stack into the CPU registers. It achieves this by omitting numerous costly memory accesses and computations within the C++ array a (slow program) and utilizes the variables s and cnt (fast program) which are stored in the CPU registers. Besides its runtime efficiency, the CODEX generation is also desirable because its sequence length and magnitudes are not bounded like the slower example. Due to the complexity of reasoning about the different data structures involved as well as the memory hierarchy, it would be difficult for static analysis techniques to yield such transformations. We also show more examples of a model-generated Python code edits in Appendix B.

While some of these optimizations may seem conceptually simple, they are impractical or non-trivial to try to implement via static analysis given the necessity to *reason* about loops, memory accesses, libraries, and transformations that may even change the behavior of the code over edge-cases. Search procedures may also struggle to identify such optimizations, because the search space for such large transformations may be unwieldy. Finally, text-based explanations can help clarify and give context to the optimizations performed by models trained using PIE. Our results show that LLMs can explain the reasoning behind these optimizations when provided with a few examples. In fact, the explanation in Figure 1 was generated using a few-shot prompting technique, which is described in more detail in Appendix C. Additionally, we analyze how problem difficulty affects accuracy and solve rate in Appendix D.

[6]The sequence from the Collatz Conjecture. It is formed by repeatedly applying the following procedure: "*if the number is even, divide it by two, otherwise, multiply it by three and add one.*"

Comparing CODEX and CODEGEN. We analyze the programs optimized by CODEX and CODEGEN-16B-mono for the 32 sample setting for Python . Out of 1000 programs in the test set, CODEX optimized 436 programs, while CODEGEN optimized 303. There were 229 programs that both models optimized. Interestingly, in these 229 cases, there was no statistically significant difference in the length of the generated solution (236 for CODEX vs. 244 for CODEGEN) and the average speedup (2.75 for CODEX vs. 2.73 for CODEGEN). The speedups were also highly correlated (corr. coeff. 0.71). Despite these similarities, CODEX modified the code to a higher degree, with a statistically significantly higher average diff length (70% of the code vs. 56% for CODEX). On inspecting the output, we found that CODEX tends to modify larger portions of the program and often completely changes the core algorithm. This may not be desirable from a usability perspective because programmers may have to spend more time understanding the changes made. We also observed that CODEX sometimes renames all the variables in the given program (e.g., setting a, b, c to x, y, z). On the other hand, CODEGEN tends to perform small, targeted changes. This suggests an additional benefit of finetuning — it may be helping CODEGEN in learning to perform minimal edits and preserving the variable names. We include qualitative examples in Appendix G.

Possibility of spurious optimizations Upon analysis, we find a few cases where CODEGEN tries to reduce runtime by setting constants to lower values. Common examples include the number of iterations and the size of data structures. In general, such changes are undesirable since they make strong assumptions about the range of inputs. While it is easy to revert such changes by performing static analysis, these cases highlight the need to perform thorough testing by including a diverse set of test cases.

6 RELATED WORK

Below, we review the most relevant work in machine learning for optimization. In Appendix I we discuss related work in program synthesis and code generations with language models.

Optimization using Machine Learning. Using machine learning algorithms and heuristics (Petke et al., 2017) for compiler optimization has been explored extensively in code refactoring (Mens & Tourwé, 2004; Agnihotri & Chug, 2020), compiler transformation (Bacon et al., 1994; Talaashrafi, 2022), parameter search (Hamadi & Hamadi, 2013; Huang et al., 2019; Kaufman et al., 2021), auto-vectorization (Nuzman et al., 2006; Mendis et al., 2019), GPU code optimization (Liou et al., 2020; Cummins et al., 2021), automatic algorithm selection (Kotthoff, 2016; Kerschke et al., 2019), and room at the top (Leiserson et al., 2020; Sherry & Thompson, 2021). DeepPERF (Garg et al., 2022) uses a transformer-based model fine-tuned for the task of generating performance improvement patches for C# applications. MAGPIE (Blot & Petke, 2022) is a unified software performance improvement framework that enables various performance improvement techniques such as compiler optimizations, algorithm parameter tuning, and evolving source code with genetic algorithm. SCALENE (Berger et al., 2022) designs a low-overhead Python-profiler that helps the developers to identify inefficiencies in their Python code and apply relevant optimization. While the main body of the paper primarily uses a sampling-based approach for profiling, the GitHub code of SCALENE[7] demonstrates few examples of using OpenAI CODEX for suggesting Python optimizations. PIE, on the other hand, uses large language models and smaller-sized code models fine-tuned on our performance dataset to suggest functionally correct performance-improving code edits in C++ and Python applications.

7 DISCUSSION AND FUTURE WORK

This work explores the capability of language models of code in generating performance-improving edits, while adhering to the functional correctness constraint. Our results demonstrate that such models can achieve speedup

[7]https://github.com/plasma-umass/scalene

up to $2.5\times$ for over 25% of test programs in C++ and Python . This research is the initial step towards unlocking the potential of language models in leveraging the opportunities at the "top" of the computing stack. Particularly, we improve algorithmic efficiency and enable automatic code optimization without taking additional cycles from developers. Our promising results and the proven capability of large language models in a variety of tasks sketch an exciting path forward to drive the computing efficiency in the post-Moore era. Nonetheless, there are many research directions and challenges that need attention, such as automatic test generation for performance triage, improving the alignment of generated code edits to target system and/or user preferences, adding structure to generated code edits, and tailoring code edit generation to architecture and hardware features.

ACKNOWLEDGEMENTS

This material is partly based on research sponsored in part by the Air Force Research Laboratory (agreement number FA8750-19-2-0200). The U.S. Govt. is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Research Laboratory or the U.S. Government. We would like to extend our gratitude towards Herman Schmit, Chandu Thekkath, James Laudon, and Stella Aslibekyan for reviewing the paper and providing insightful feedback. We also thank the extended team at Google Research, Brain Team who enabled this research and helped us with the paper.

REFERENCES

Hojjat Aghakhani, Wei Dai, Andre Manoel, Xavier Fernandes, Anant Kharkar, Christopher Kruegel, Giovanni Vigna, David Evans, Ben Zorn, and Robert Sim. TrojanPuzzle: Covertly Poisoning Code-Suggestion Models. *arXiv preprint arXiv:2301.02344*, 2023.

Mansi Agnihotri and Anuradha Chug. A Systematic Literature Survey of Software Metrics, Code Smells and Refactoring Techniques. *Journal of Information Processing Systems*, 2020.

Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. *Compilers: Principles, Techniques, and Tools*, volume 2. Addison-wesley Reading, 2007.

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 preprint arXiv:2108.07732*, 2021.

David F Bacon, Susan L Graham, and Oliver J Sharp. Compiler Transformations for High-Performance Computing. *CSUR*, 1994.

Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. DeepCoder: Learning to Write Programs. *arXiv preprint arXiv:1611.01989*, 2016.

Emery D Berger, Sam Stern, and Juan Altmayer Pizzorno. Triangulating Python Performance Issues with SCALENE. *arXiv preprint arXiv:2212.07597*, 2022.

Sid Black, Leo Gao, Phil Wang, Connor Leahy, and Stella Biderman. GPT-Neo: Large Scale Autoregressive Language Modeling with Mesh-Tensorflow, 2021.

Aymeric Blot and Justyna Petke. MAGPIE: Machine Automated General Performance Improvement via Evolution of Software, 2022.

Rudy Bunel, Matthew Hausknecht, Jacob Devlin, Rishabh Singh, and Pushmeet Kohli. Leveraging Grammar and Reinforcement Learning for Neural Program Synthesis. *arXiv preprint arXiv:1805.04276*, 2018.

José Cambronero, Sumit Gulwani, Vu Le, Daniel Perelman, Arjun Radhakrishna, Clint Simon, and Ashish Tiwari. FlashFill++: Scaling Programming by Example by Cutting to the Chase. *POPL*, 2023.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. CODET: Code Generation with Generated Tests. *arXiv preprint arXiv:2207.10397*, 2022a.

Binghong Chen, Daniel Tarlow, Kevin Swersky, Martin Maas, Pablo Heiber, Ashish Naik, Milad Hashemi, and Parthasarathy Ranganathan. Learning to Improve Code Efficiency, 2022b.

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 preprint arXiv:2107.03374*, 2021.

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

Jacob Devlin, Rudy R Bunel, Rishabh Singh, Matthew Hausknecht, and Pushmeet Kohli. Neural Program Meta-Induction. *NeurIPS*, 2017a.

Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. RobustFill: Neural Program Learning under Noisy I/O. In *ICML*, 2017b.

William Falcon and The PyTorch Lightning team. PyTorch Lightning. *10.5281/zenodo.3828935*, 3 2019. URL `https://github.com/Lightning-AI/lightning`.

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 preprint arXiv:2204.05999*, 2022.

Pranav Garg and Srinivasan H Sengamedu. Synthesizing Code Quality Rules from Examples. *OOPSLA*, 2022.

Spandan Garg, Roshanak Zilouchian Moghaddam, Colin B. Clement, Neel Sundaresan, and Chen Wu. DeepPERF: A Deep Learning-Based Approach For Improving Software Performance, 2022.

Youssef Hamadi and Youssef Hamadi. Autonomous Search. *Combinatorial Search: From Algorithms to Systems*, 2013.

Changwu Huang, Yuanxiang Li, and Xin Yao. A Survey of Automatic Parameter Tuning Methods for Metaheuristics. *IEEE transactions on evolutionary computation*, 2019.

Naman Jain, Skanda Vaidyanath, Arun Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram Rajamani, and Rahul Sharma. Jigsaw: Large Language Models Meet Program Synthesis. In *ICSE*, 2022.

Ashwin Kalyan, Abhishek Mohta, Oleksandr Polozov, Dhruv Batra, Prateek Jain, and Sumit Gulwani. Neural-guided Deductive Search for Real-time Program Synthesis from Examples. *arXiv preprint arXiv:1804.01186*, 2018.

Sam Kaufman, Phitchaya Phothilimthana, Yanqi Zhou, Charith Mendis, Sudip Roy, Amit Sabne, and Mike Burrows. A Learned Performance Model for Tensor Processing Units. *Proceedings of Machine Learning and Systems*, 2021.

Pascal Kerschke, Holger H Hoos, Frank Neumann, and Heike Trautmann. Automated Algorithm Selection: Survey and Perspectives. *Evolutionary computation*, 2019.

Lars Kotthoff. Algorithm Selection for Combinatorial Search Problems: A Survey. *Data mining and constraint programming: Foundations of a cross-disciplinary approach*, 2016.

Charles E Leiserson, Neil C Thompson, Joel S Emer, Bradley C Kuszmaul, Butler W Lampson, Daniel Sanchez, and Tao B Schardl. There's Plenty of Room at the Top: What Will Drive Computer Performance After Moore's Law? *Science*, 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*, 2022.

Jhe-Yu Liou, Xiaodong Wang, Stephanie Forrest, and Carole-Jean Wu. GEVO: GPU Code Optimization using Evolutionary Computation. *TACO*, 2020.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. *arXiv preprint arXiv:2102.04664*, 2021.

Charith Mendis, Cambridge Yang, Yewen Pu, Dr Amarasinghe, Michael Carbin, et al. Compiler Auto-Vectorization with Imitation Learning. *Advances in Neural Information Processing Systems*, 2019.

Na Meng, Miryung Kim, and Kathryn S McKinley. Systematic Editing: Generating Program Transformations from an Example. *ACM SIGPLAN Notices*, 2011.

Tom Mens and Tom Tourwé. A Survey of Software Refactoring. *IEEE Transactions on software engineering*, 2004.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. CodeGen: An Open Large Language Model for Code with Multi-turn Program Synthesis. *arXiv preprint arXiv:2203.13474*, 2022.

Dorit Nuzman, Ira Rosen, and Ayal Zaks. Auto-vectorization of Interleaved Data for SIMD. *ACM SIGPLAN Notices*, 2006.

Maxwell Nye, Luke Hewitt, Joshua Tenenbaum, and Armando Solar-Lezama. Learning to Infer Program Sketches. In *ICML*, 2019.

Augustus Odena and Charles Sutton. Learning to Represent Programs with Property Signatures. *arXiv preprint arXiv:2002.09030*, 2020.

Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. Neuro-symbolic Program Synthesis. *arXiv preprint arXiv:1611.01855*, 2016.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic Differentiation in Pytorch. *NeurIPS Workshop Autodiff*, 2017.

Neil Perry, Megha Srivastava, Deepak Kumar, and Dan Boneh. Do Users Write More Insecure Code with AI Assistants? *arXiv preprint arXiv:2211.03622*, 2022.

Justyna Petke, Saemundur O Haraldsson, Mark Harman, William B Langdon, David R White, and John R Woodward. Genetic Improvement of Software: A Comprehensive Survey. *IEEE Transactions on Evolutionary Computation*, 2017.

Ruchir Puri, David S. Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, Veronika Thost, Luca Buratti, Saurabh Pujar, Shyam Ramji, Ulrich Finkler, Susan Malaika, and Frederick Reiss. CodeNet: A Large-Scale AI for Code Dataset for Learning a Diversity of Coding Tasks, 2021.

Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 3505–3506, 2020.

Raimondas Sasnauskas, Yang Chen, Peter Collingbourne, Jeroen Ketema, Gratian Lup, Jubi Taneja, and John Regehr. Souper: A Synthesizing Superoptimizer. *arXiv preprint arXiv:1711.04422*, 2017.

Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic Superoptimization. *ACM SIGARCH Computer Architecture News*, 41(1):305–316, 2013.

Yash Sherry and Neil C. Thompson. How Fast Do Algorithms Improve? [Point of View]. *Proceedings of the IEEE*, 2021.

Hui Shi and Yang Zhang. Deep Symbolic Superoptimization without Human Knowledge. *ICLR 2020*, 2020.

Alex Shypula, Pengcheng Yin, Jeremy Lacomis, Claire Le Goues, Edward Schwartz, and Graham Neubig. Learning to Superoptimize Real-world Programs. *arXiv preprint arXiv:2109.13498*, 2021.

Delaram Talaashrafi. *Advances in the Automatic Detection of Optimization Opportunities in Computer Programs*. PhD thesis, Western University, 2022.

Georgios Tournavitis, Zheng Wang, Björn Franke, and Michael FP O'Boyle. Towards a Holistic Approach to Auto-parallelization: Integrating Profile-driven Parallelism Detection and Machine-learning Based Mapping. *ACM Sigplan notices*, 2009.

Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. Unit Test Case Generation with Transformers and Focal Context. *arXiv preprint arXiv:2009.05617*, 2020.

Lewis Tunstall, Leandro Von Werra, and Thomas Wolf. *Natural Language Processing with Transformers*. "O'Reilly Media, Inc.", 2022.

Ben Wang and Aran Komatsuzaki. GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model. https://github.com/kingoflolz/mesh-transformer-jax, May 2021.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 conference on empirical methods in natural language processing: system demonstrations*, pp. 38–45, 2020.

Yingfei Xiong and Bo Wang. L2S: A Framework for Synthesizing the Most Probable Program under a Specification. *TOSEM*, 2022.

Frank F Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. A Systematic Evaluation of Large Language Models of Code. In *MAPS*, 2022.

Michihiro Yasunaga and Percy Liang. Graph-based, Self-supervised Program Repair from Diagnostic Feedback. In *ICML*, 2020.

Shuyan Zhou, Uri Alon, Frank F Xu, Zhengbao JIang, and Graham Neubig. DocPrompting: Generating Code by Retrieving and Reading Docs. *arXiv preprint arXiv:2207.05987*, 2022.

APPENDIX

A LIMITATIONS

Limited study of programming languages. This work primarily evaluates the capability of code models in suggesting performance improving code edits for two popular programming languages, Python and C++. This work does not claim the generality of these models in generating performance improving code edits for other programming languages. Nonetheless, we speculate that with sufficient data and training, code models can potentially unlock similar capabilities for other programming languages.

Auto-parallelization. Recent work has explored various auto-parallelization methods Tournavitis et al. (2009); Nuzman et al. (2006); Mendis et al. (2019), whereas this work does not particularly focus on any specific parallelization method. While we have not observed code edits for parallelization, which we suspect is a repercussion of training dataset, we surmise code models can be repurposed for specialized parallelization code edits. Evaluating code models tailored for auto-parallization is an interesting research avenue that we leave as future work.

Security vulnerability. Recent studies Perry et al. (2022); Aghakhani et al. (2023) raised concerns about security vulnerability of AI-assisted code generation. This work neither assess nor quantify the probable security vulnerability in the generated code from code models. Nonetheless, we recognize and value these concerns and do not recommend using the generated code in products.

B ADDITIONAL MODEL-GENERATED EXAMPLES

B.1 EXAMPLE 1

Figure 4 is an example of a program that calculates the position of the right-most bit in a bit-vector for the smallest element in a list. We see numerous changes in the CODEX optimized version. [1] and import is

```
import numpy as np
x = int(eval(input()))
y = input().split()
y = [int(i) for i in y]
y = np.array(y)
count = 0

while sum([y[i] % 2 \
      for i in range(len(y))]) == 0:
    y = y/2
    count += 1
print(count)
```

(a) Slower Code.

```
N = int(input())
A = list(map(int, input().split()))
count = 0

while all(a % 2 == 0 for a in A):
    A = [a // 2 for a in A]
    count += 1
print(count)
```

(b) Faster Code

Figure 4: Example of slow code (above) and faster version generated by CODEX.

removed and list-comprehensions are used instead, [2] the loop guard no longer indexes into an array, and [3] the use of primitive integers/integer division. After profiling the numerous optimizations, we found that removing the import was responsible for 95% of the speedup; however, after removing the import, the change to the loop guard still caused another relative 72% decrease in runtime. All remaining changes were responsible for the a remaining 22% decrease reduction in runtime.

B.2 EXAMPLE 2

```
n = int(eval(input()))
l = []
for i in range(1, 1000001):
    a = n / i
    if a.is_integer():
        l.append(i + n / i)
print((int(min(l) - 2)))
```

(a) Slower Code.

```
n = int(input())
ans = float("inf")
for i in range(1, int(n ** 0.5) + 1):
    if n % i == 0:
        ans = min(ans, i + n // i - 2)
print(int(ans))
```

(b) Faster Code

Figure 5: Example of slow code (above) and faster version generated by CODEX for the minimum sum of two factors of a positive integer n.

Figure 5 calculates the minimum sum of two factors of a positive integer n. The first version is less optimized compared to the second one as it uses a loop to check every number from 1 to 1000001 (max possible value of n), dividing n by each of them to see if the result is an integer. If so, the sum of that number and n divided by that number is appended to the list l. After the loop, it calculates the minimum value in l and subtracts two from it. The second version is optimized as it only checks numbers up to the square root of n plus one. It uses a variable ans initialized with positive infinity and updates it with the minimum sum of two factors of n found in the loop. Both versions produce the same output, but the second one is faster for large values of n.

C GENERATING NATURAL LANGUAGE EXPLANATIONS FOR CODE EDITS

Automated code optimization techniques aim to improve the performance of code, but there are several open issues with such an approach. Even though changes at the high-level code are more straightforward to understand than those at low-levels, some edits may be obscure and hard to understand, leading to low confidence in accepting them by developers. Moreover, some edits may also be incorrect. Additionally, proposing edits without a proper explanation may not aid in the personal development of programmers and may encourage over-reliance on assisted tools.

To address these issues, text-based explanations may provide clarity and context to the optimization types and help mitigate these challenges. Specifically, we create two demonstrations of (slow program, fast program, explanatory text for optimization type). These demonstrations are used to create a prompt, which is used to generate explanations for the target (slow, fast) pair, shown in Figure 6 and Figure 7 for Python and C++ , respectively. As these code examples demonstrate, CODEX is capable of not only identifying the faster version of the code, but also provide an accurate explanation for the optimization types (use of the builtin function `min` which is written in low-level language).

```python
A = list(map(int, input().split()))
B = list(map(int, input().split()))

is_lower = True
for i in range(len(A)):
    for j in range(len(B)):
        if A[i] > B[j]:
            is_lower = False

if is_lower:
    print("Yes")
else:
    print("No")
```

(a) Slower Code.

```python
A = list(map(int, input().split()))
B = list(map(int, input().split()))

if max(A) <= min(B):
    print("Yes")
else:
    print("No")
```

(b) Faster Code

```
The first version is using two nested loops to compare each element in A to
    each element in B, which takes O(N^2) time. The second version is using
    two built-in functions (max and min) to compare the maximum value in A to
    the minimum value in B, which takes O(N) time. Therefore, the second
    version is faster as it reduces the time complexity of the code.
```

(c) Explanation

Figure 6: While PIE optimizes programs, the proposed algorithmic changes can be non-obvious. The illustrative example above shows slower and faster code, and a CODEX generated natural language explanation of the reason of optimization. The fact that CODEX can generate such explanations is promising, not only for making the process more transparent, but also for leveraging these explanations for improving the performance.

D ADDITIONAL ANALYSIS

We analyze the effect of program length on the accuracy of the generated programs, and the relative runtime reduction (RTR). The accuracy and RTR metrics are shown in Figure 8. As the program length increases, the accuracy generally decreases. As expected, smaller programs (less than 16 LOC) are solved with perfect accuracy. Similarly, the RTR of programs of length 16 is 0, indicating that there are fewer opportunities to optimize for these smaller programs. The RTR values increased as program length increased from 16 to 128, reaching a maximum value of 28.16 for a program length of 128. After this, the RTR values decreases with the program length. We hypothesize that both programs at either end of the LOC are not amenable to our method: short programs may not have much room to improve further, whereas language models might struggle with optimizing longer programs, an interesting avenue for future research.

E FEW-SHOT PROMPTS EXAMPLES

Figure 10 and Figure 11 depict one example from our eight-shot learning method in CODEX to generate explanation for the optimization type. Figure 12 and Figure 13 demonstrate one example from two-shot learning method in CODEX to generate explanation for the optimization type.

```cpp
#include<iostream>
#include<string>
using namespace std;
int main(){
    string s;
    int k;
    cin>>k>>s;
    if(s.length()>k){
        for(int i=0;i<k;i++){
            cout<<s[i];
        }
        cout<<"..."<<endl;
    }
    else cout<<s<<endl;
}
```

(a) Slower Code.

```cpp
#include<bits/stdc++.h>
using namespace std;
int main(){
    string s;
    int k;
    cin>>k>>s;
    if(s.length()>k)

    cout<<s.substr(0,k)<<"..."<<endl;
    else cout<<s<<endl;
}
```

(b) Faster Code

```
The second version is optimized because it uses string::substr() to get the
    first k characters of the string instead of iterating through the string
    to get the k characters. This is more efficient because substr() has been
    optimized for extracting substrings from strings.
```

(c) Explanation

Figure 7: While PIE optimizes programs, the proposed algorithmic changes can be non-obvious. The example above shows slower and faster code, and a CODEX generated natural language explanation of the reason of optimization. The fact that CODEX can generate such explanations is promising, not only for making the process more transparent, but also for leveraging these explanations for improving the performance.

F TRAINING DETAILS

We retrieve the pre-trained checkpoint models from Nijkamp et al. (2022) through HuggingFace (Wolf et al., 2020), and further tune them using PyTorch (Paszke et al., 2017), PyTorch Lightning (Falcon & The PyTorch Lightning team, 2019), and DeepSpeed (Rasley et al., 2020). We do not extensively modify the hyperparameters, and use the values specified by Nijkamp et al. (2022) for each model.

G CODEX VS. CODEGEN

Figure 14, Figure 16, and Figure 15 provide some qualitative examples of differences between optimizations performed by CODEX and CODEGEN-16B-mono in 32 sample setting.

H PIE DATASET METADATA

In the PIE dataset, each submission for a given problem description contains the information in Table 5.

I ADDITIONAL RELATED WORK

Code generation with language models. Recent work has explored the capabilities of large language models in various code generation applications. AlphaCode (Li et al., 2022) leverages language models to generate high-

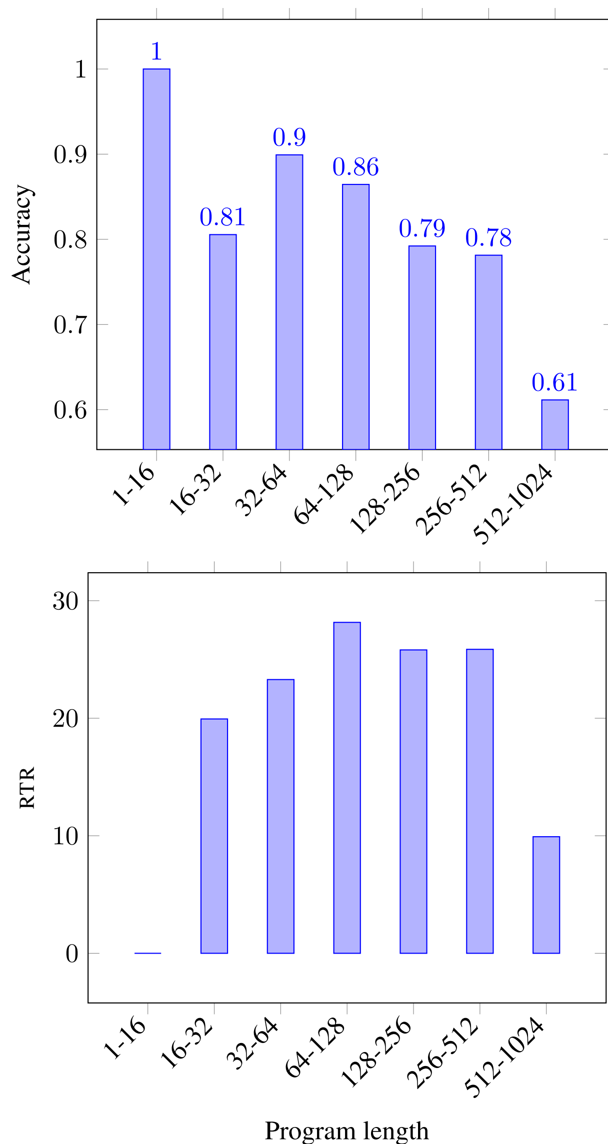

Figure 8: Histograms of accuracy and RTR for different program lengths measured in program token length determined by the Python lexical scanner in the TOKENIZE library. Program lengths roughly correlate with complexity: challenging problems require larger programs. The program lengths are indicated along the x-axis. The labels indicate the maximum LOC for each bin (e.g., the first bin shows programs that have $\leqslant 16$ LOC). Accuracy of the generated programs decreases with complexity, as expected. Interestingly, both very long and very short programs do not gain much from 🥧. We hypothesize that this is due to lack of opportunities and the reduction in runtime

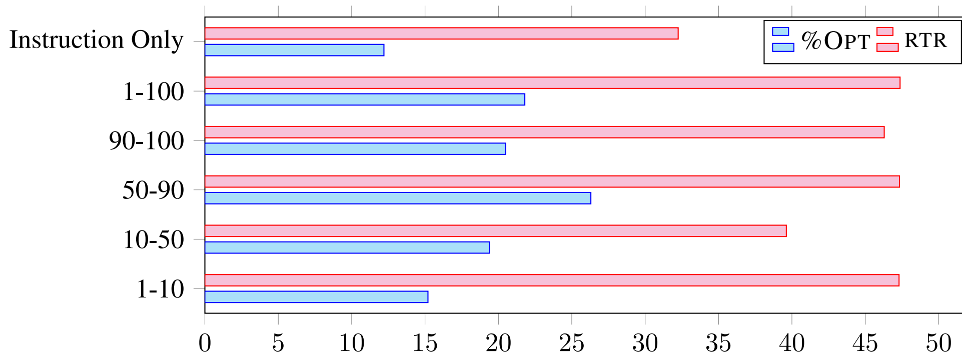

Figure 9: Analysis of the prompt type on the final performance. The "Instruction Only" bar shows the results when the prompt merely instructs the model to generate an optimized program version. The instruction is in the form of: "# Optimized version of the program:". On average, the "instruction only" prompt improves ⌈%OPT⌋ by $\approx 13\%$, whereas using demonstrations of slow-fast pairs leads to up to $\approx 26\%$ improvement.

Option	Value
Learning Rate	2e-6
Number of GPUs	4 × NVIDIA A6000
Warmup Proportion	0.05
Evaluation Batch Size	1
Input/Output Sequence length	300
Model Name or Path	Salesforce/codegen-16B (mono for Python , multi for C++)
Seed	10708
Validation Metric	Loss
Save Top K	1
Accumulate Gradient Batches	2
Strategy	deepspeed_stage_2_offload
Accelerator	GPU
Precision	16
Training steps	5000
Gradient Clip Value	1.0

Table 3: Training Configuration for the 16B models. The training took about 3 days for both the models.

quality code and unit-tests comparable with human competitors. OpenAI Codex (Chen et al., 2021) suggests code snippets to software developers on-the-fly, thereby increasing their productivity. CodeT (Chen et al., 2022a) takes an alternative direction in automating the test generation process for code samples, which facilitates the evaluation of the quality and correctness of generated code samples. In a similar vein, the community has observed a surge in open source language models for code generation (Black et al., 2021; Wang & Komatsuzaki, 2021; Tunstall et al., 2022; Xu et al., 2022; Fried et al., 2022; Nijkamp et al., 2022; Tufano et al., 2020), each targeting to enhance different part of software developing experience. While we take a similar approach in using large language models for code generation, we alternatively explore and demonstrate the capability of these models in suggesting high performance code edits.

Program synthesis. Program synthesis encompasses methods that construct programs from high-level formal specifications. Recent work leverages machine learning to assist program synthesis (Balog et al., 2016; Bunel

```
# slower version:

N, M=map(int, input().split())
import numpy as np

pena=np.array([0]*N)
status=np.array([0]*N)


for i in range(M):
  p,s=tuple(input().split())
  p=int(p)
  if s=='WA':
    if status[p-1]!=1:
      pena[p-1]+=1
  if s=='AC':
    if status[p-1]!=1:
      status[p-1]=1
nac=sum(status)
npena=sum(pena*status)
print(nac, npena)
```

(a) Slower Code

```
# optimized version of the same code:

N,M=map(int, input().split())
now=[0]*N
pena=0
for i in range(M):
    num,result=input().split()
    num=int(num)-1
    if result=='AC':
        if now[num]>0:
            pena+=now[num]
        now[num]=-1
    else:
        if now[num]>=0:
            now[num]+=1
print(now.count(-1), pena)
```

(b) Faster Code (50% faster)

Figure 10: An example of prompt used in eight-shot learning for optimizing Python code.

et al., 2018; Devlin et al., 2017a;b; Kalyan et al., 2018; Parisotto et al., 2016; Odena & Sutton, 2020; Nye et al., 2019; Meng et al., 2011; Jain et al., 2022; Austin et al., 2021; Xiong & Wang, 2022; Cambronero et al., 2023; Garg & Sengamedu, 2022; Yasunaga & Liang, 2020), exclusively satisfying high-level constraints and specifications and overlook the efficiency of the generated code.

Optimization using machine learning. Recently, Chen et al. (2022b) uses discrete variational auto-encoder, each latent representation maps to a different category of code edits, and canonicalized code representations to automatically suggest Python *hints* to improve the code efficiency. In contrast, PIE directly leverages large language models to spit out functionally correct code with increased performance across multiple programming

```
# slower version:

#include <iostream>
#include<vector>
#include<algorithm>
#include<string>
#include<cmath>
#include<set>
using namespace std;
int main()
{
        int M1,D1;
        cin >> M1 >> D1;
        int M2, D2;
        cin >> M2 >> D2;
        if (M1 + 1 == M2 && D2 == 1) {
                cout << 1 << endl;
        }
        else {
                cout << 0 << endl;
        }

}
```

(a) Slower Code

```
# optimized version of the same code:

#include <iostream>
#include<vector>
#include<algorithm>
#include<string>
#include<cmath>
#include<set>
using namespace std;
int main()
{
        int M1, D1, M2, D2;
        cin >> M1 >> D1 >> M2 >> D2;
        if (M1 + 1 == M2 && D2 == 1) {
                cout << 1 << endl;
        }
        else {
                cout << 0 << endl;
        }
}
```

(b) Faster Code (50% faster)

Figure 11: An example of prompt used in eight-shot learning for optimizing C++ code.

languages. Additional work has leveraged search-based (Schkufza et al., 2013), constraint-solving (Sasnauskas et al., 2017), and deep learning (Shypula et al., 2021) to superoptimize programs at the level of assembly

```python
# slower version:
S = eval(input())
T = eval(input())
flag = 0
for i in range(len(S)):
    if S[i] != T[i]:
        flag = 1
        break
if flag == 1:
    print('No')
else:
    print('Yes')
```

(a) Slower Code.

```python
# optimized version of the same code:
S = eval(input())
T = eval(input())
if S == T[0 : len(S)]:
    print('Yes')
else:
    print('No')
```

(b) Faster Code

```
# explanation for why the second version is faster:

The second version is faster because it uses a direct comparison of the two
    strings (S and T[0 : len(S)]) instead of looping through each element and
    checking for differences. This reduces the time complexity of the code,
    making it faster. The direct comparison is more efficient as it takes
    advantage of the built-in string comparison functionality in Python.
```

(c) Explanation

Figure 12: An example of prompt used in two-shot learning for generating the explanation for optimization type in Python code.

Option	Value
Learning Rate	5e-6
Number of GPUs	2× NVIDIA A6000
Training Batch Size	6
Evaluation Batch Size	4
Model Name or Path	Salesforce/codegen-2B (mono for Python , multi for C++)

Table 4: Training Configuration for the 2B models. Only the parameters that differ from the 16B setup are shown here. The training took about 1 day for both the models.

or compiler intermediate representations. Unlike our work, they all operate on low-level programs usually obfuscated from the programmer and rely on simplified proxies for program runtime.

```
# slower version:

#include<bits/stdc++.h>
using namespace std;
int main() {
    int n;
    cin >> n;
    int64_t ans = 0;
    for (int i = 1; i < n; i++) ans += i;
    cout << ans << endl;
}
```

(a) Slower Code.

```
# optimized version of the same code:

#include<bits/stdc++.h>
using namespace std;
int main() {
    int64_t n;
    cin >> n;
    cout << n*(n-1)/2 << endl;
}
```

(b) Faster Code

```
# explanation for why the second version is faster:

The second version is optimized because it uses a mathematical formula
    (n*(n-1)/2) to calculate the sum of the integers from 1 to n-1 instead of
    using a loop to iterate through the numbers and add them one by one. This
    formula directly calculates the sum, which is faster than using a loop.
```

(c) Explanation

Figure 13: An example of prompt used in two-shot learning for generating the explanation for optimization type in C++ code.

Name of Column	**Description**
submission_id	Submission identifier
problem_id	Problem identifier
user_id	User identifier
date	Unix timestamp of the submission
language	mapped language of the submission (ex: C++ $\rightarrow$ C++)
cpu_time	execution time
memory	memory used

Table 5: Each submission in CodeNet is annotated with metadata listed above, enabling us to create trajectories of programs created by the same programmer for the same problem.

```python
import numpy as np
n = int(eval(input()))
a = np.array(input().split(), np.int)
a.sort()
ans = []
for x in range(a[0], a[-1] + 1):
    ans.append(np.sum((a - x) ** 2))
print((min(ans)))
```

(a) Input Code.

```python
n = int(input())
a = sorted(map(int, input().split()))
ans = 10**18
for x in range(a[0], a[-1] + 1):
    ans = min(ans, sum((y - x) ** 2 for y in a))
print(ans)
```

(b) CODEX generated code (Speedup: 3.17×)

```python
n = int(eval(input()))
a = [int(s) for s in input().split()]
mi = 10**20
for x in range(min(a), max(a) + 1):
    res = 0
    for i in range(n):
        res += (a[i] - x) ** 2
    if mi > res:
        mi = res
print(mi)
```

(c) CODEGEN-16B-mono generated (Speedup: 5.5×)

Figure 14: Example of an input code (above) and two generated solutions using Codex and Codegen. Both solutions produce the same output, but the one generated by CODEGEN-16B-mono is faster as it avoids sorting

```
num = int(input())
lst = []
for i in range(1, num + 1):
    if num % i == 0:
        lst.append(i)
print(sum(lst) - num)
```

(a) Original Code.

```
n = int(input())
ans = 0
for i in range(1, n + 1):
    if n % i == 0:
        ans += i
print(ans - n)
```

(b) Code Generated by CODEX (Speedup: 1.77×).

Figure 15: Example of original code (above) and code generated by CODEX. CODEX tends to change while generating solutions even when such changes are not necessary. These cosmetic changes lead to larger diffs. Here, CODEX changes $x \rightarrow N$, and $y \rightarrow A$.

```python
import numpy as np

cookies = np.array([int(x) for x in input().split()])
if any(cookies % 2):
    ans = 0
elif cookies[0] == cookies[1] == cookies[2]:
    ans = -1
else:
    ans = 0
    while True:
        if not any(cookies % 2):
            cookies = cookies // 2
            ans += 1
            continue
        if all(cookies % 2):
            ans += 1
        break
print(ans)
```

(a) Input code.

```python
A, B, C = map(int, input().split())
ans = 0
while A % 2 == B % 2 == C % 2 == 0:
    if A == B == C:
        ans = -1
        break
    A, B, C = (B + C) // 2, (A + C) // 2, (A + B) // 2
    ans += 1
print(ans)
```

(b) CODEX generated code (Speedup: 3.5×).

```python
import numpy as np

cookies = np.array([int(x) for x in input().split()])
if cookies.sum() % 2:
    ans = 0
elif cookies[0] == cookies[1] == cookies[2]:
    ans = -1
else:
    ans = 0
    while True:
        if not any(cookies % 2):
            cookies = cookies // 2
            ans += 1
            continue
        if all(cookies % 2):
            ans += 1
        break
print(ans)
```

(c) CODEGEN-16B-mono generated code (Speedup: 1.01×).

Figure 16: Example of input code (above) and two versions of optimized code generated by CODEX and CODEGEN, respectively. Both versions are equivalent, but the version generated by CODEX completely changes the algorithm.