Speed Up Your Code: Progressive Code Acceleration Through Bidirectional Tree Editing

Longhui Zhang[1], Jiahao Wang[1], Meishan Zhang[1], Gaoxiong Cao[2], Ensheng Shi[2], Yuchi Ma[2], Jun Yu[1], Honghai Liu[1], Jing Li[1]✉, Min Zhang[1]
[1]Harbin Institute of Technology, Shenzhen, China, [2]Huawei, Shenzhen, China
longhuizhang97@gmail.com jingli.phd@hotmail.com

Abstract

Large language models (LLMs) have made significant strides in code acceleration (CA) tasks. Current works typically fine-tune LLMs using slow-fast code pairs mined from online programming platforms. Although these methods are widely recognized for their effectiveness, the training data often lack clear code acceleration patterns and offer only limited speed improvements. Moreover, existing training methods, such as direct instruction fine-tuning (IFT), tend to overlook the hierarchical relationships among acceleration patterns. In this work, we introduce BITE, a novel training paradigm designed to improve LLMs' CA capabilities through two key innovations: (1) *Bidirectional tree editing*, which generates high-quality training data by incrementally transforming given code into both its most efficient and least efficient variants, and (2) *Progressive code acceleration learning*, which enables LLMs to internalize multi-level CA strategies by learning increasingly sophisticated acceleration patterns. Additionally, we introduce a new CA evaluation benchmark and metric for comprehensive assessment of model performance on CA tasks. Extensive experiments on both our benchmark and existing benchmarks demonstrate the effectiveness of our approach. Notably, BITE enables Qwen$_{1.5B}$ to outperform prompt-enhanced GPT-4 and current training-based methods on average across five programming languages.

1 Introduction

As outlined by the ISO/IEC 25010 software quality guidelines, computational efficiency is a critical indicator of system quality (ISO/IEC25010, 2011). Consequently, CA, the task of automatically refactoring inefficient code for speed improvement, has been an active area of research (Mankowitz et al., 2023). Early CA works (Nistor et al., 2013) usually relied on manually defined inefficiency types and corresponding modifications—referred to as *Acceleration Patterns*—to optimize the code. With the advent of deep learning, it became possible to learn from CA data, thereby expanding the scope of inefficient code that can be addressed (Garg et al., 2022). However, achieving the performance necessary for practical deployment remains a challenge.

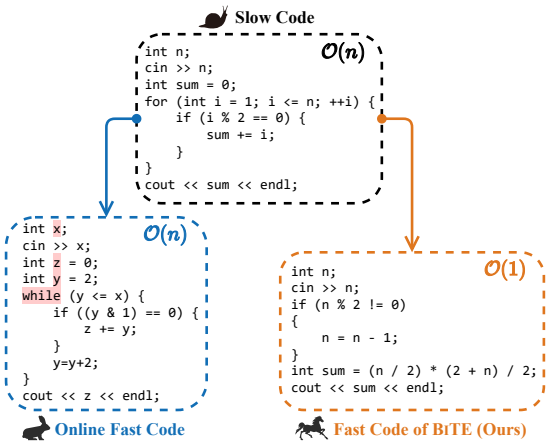

Figure 1: Limitations of using slow-fast code pairs from online platforms as training data. These data often contain task-irrelevant variations (*e.g.,* variable naming conventions or loop structure choices) that hinder models from identifying meaningful optimization patterns. Additionally, the limited speedup ratio leads the model to favor overly conservative code changes.

The exceptional performance of LLMs like GPT-4 (OpenAI, 2023) has opened new possibilities for CA tasks. Initial explorations using simple prompt-based learning have shown promise (Mankowitz et al., 2023), with further enhancements achieved through optimized prompting strategies such as in-context learning (Waghjale et al., 2024) and chain-of-thought (Gao et al., 2024). Training-based strategies have proven even more effective , particularly those that fine-tune LLMs on paired slow and fast code solutions collected from online platforms like Codeforces (Shypula et al., 2024). These methods have demonstrated superior performance compared

✉ Corresponding author.

Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 28563–28576

to traditional deep learning and prompt-based approaches (Waghjale et al., 2024).

Despite these advancements, training-based CA models still face two challenges. (1) **Obscure Acceleration Patterns and Limited Speedup in Training Data**: As shown in Figure 1, slow and fast code snippets collected from online platforms often originate from different developers, resulting in style differences (*e.g.,* variable naming) that obscure the underlying acceleration patterns. Even when snippets come from the same developer, they usually show limited speedup since developers rarely proactively optimize their own code (Charfi et al., 2010). (2) **The Neglect of Hierarchical Relationships among Acceleration Patterns in CA Training**: Code optimization occurs at multiple levels, from simple changes (*e.g.,* removing redundant computations) to more complex algorithmic optimizations, which may build upon or combine simpler optimizations (Ouni et al., 2017). Current CA training methods, such as IFT (Shypula et al., 2024), usually treat all acceleration patterns equally, leading to suboptimal performance.

To address these challenges, we propose BITE, which tackles both the training data and methodology issues. At the data level, BITE uses *bidirectional tree editing* to automatically generate high-quality CA data with clear acceleration patterns and significant speedup. This process simulates a scenario where developers iteratively modify the code to explore efficient and inefficient code versions. At the method level, we introduce *progressive code acceleration learning*, which leverages multi-level CA data from the bidirectional tree to help models gradually master increasingly sophisticated acceleration patterns while consolidating basic optimization knowledge.

In addition to the training data and methodological study, we develop a new CA benchmark and propose a metric. The benchmark encompasses five programming languages, up-to-date code collection, efficiency-sensitive code inputs, and expert-optimized fast code. Our proposed metric measures the gap between model-optimized and expert-optimized code, serving as a progress indicator for code acceleration. Extensive experiments on our benchmark and others demonstrate the effectiveness of our approach across LLMs of varying types and scales, including StarCoder$_{3B}$ and Qwen$_{1.5-3B}$. Notably, BITE enables Qwen$_{1.5B}$ to outperform prompt-enhanced GPT-4 and a variety of training-based LLMs in the CA tasks. Additionally, we conduct detailed experiments to further analyze the effectiveness of our approach.

We summarize the key contributions of our work as follows:

- We propose BITE to automate the generation of high-quality CA data and enable LLMs to fully adapt to CA tasks through progressive learning.
- We create a new evaluation benchmark and metric to evaluate the performance of models in automatically accelerating code.
- BITE significantly improves the performance of LLMs of different sizes and types on CA tasks across five programming languages.

2 Background

Instruction Fine-tuning (IFT). IFT (Zhang et al., 2024) is a classic fine-tuning paradigm for LLMs, aimed at enhancing their ability to follow instructions and enabling efficient adaptation to a variety of tasks. This approach involves training the LLM on instruction-output pairs, with the objective of minimizing the discrepancy between the model's predicted output and the ground-truth sequence through a next-token prediction task. The training objective is expressed as follows:

$$\mathcal{L}_{\text{ift}}(in, out) = -\sum_{i} \log p\left(out_i \mid in, out_{<i}\right), \tag{1}$$

where in is the instruction input to the LLMs, out denotes the corresponding output, out_i stands for the i-th token of out, and $out_{<i}$ represents the sequence of tokens preceding the i-th token in out.

Typical Code Acceleration Learning. A common approach to adapting LLMs to CA tasks (Shypula et al., 2024; Ye et al., 2024) involves collecting code pairs that exhibit the most significant speedup for the same programming problem from online coding platforms. These pairs are then used to fine-tune the LLMs using Eq. 1. Platforms like Codeforces, which host numerous programming problems along with test cases and code solutions submitted by developers worldwide, make this approach feasible.

3 Methodology

Our BITE consists of three stages, as shown in Figure 2. We first train bidirectional code editors,

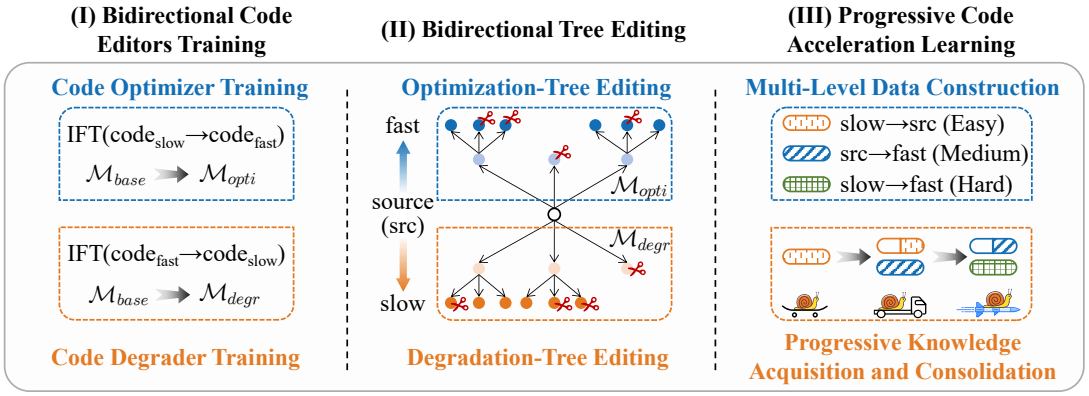

Figure 2: Overview of our BITE approach. BITE employs the trained bidirectional code editors for bidirectional tree editing, exploring both optimized and degraded code variants. Using the multi-level data from the bidirectional tree, LLMs undergo a three-stage learning pipeline to progressively master acceleration patterns of varying complexity.

which include both a code optimizer and a code degrader. These two editors are then applied in bidirectional tree editing to generate the corresponding optimized and degraded trees for the given code snippets. Finally, we employ progressive code acceleration learning to leverage multi-level training data derived from bidirectional trees for progressive knowledge acquisition and consolidation.

3.1 Bidirectional Code Editors Training

Our bidirectional editor consists of two components: the code optimizer, which is designed to enhance code efficiency, and the code degrader, which intentionally degrades code performance.

Code Optimizer Training. The training of the code optimizer $\mathcal{M}_{opti}$ follows the traditional approach outlined in Section 2, where code pairs from online programming platforms are used to fine-tune the base LLM $\mathcal{M}_{base}$. Specifically, for a slow code snippet $code_{slow}$, and a fast code snippet $code_{fast}$, we provide a code optimization prompt $\mathcal{P}_{opti}(code_{slow})$ to $\mathcal{M}_{base}$ and minimize the IFT loss $\mathcal{L}_{\text{ift}}(\mathcal{P}_{\text{opti}}(code_{slow}), code_{fast})$.

Code Degrader Training. The training process for the code degrader $\mathcal{M}_{degr}$ is identical to that of the optimizer $\mathcal{M}_{opti}$, but with the reversed input-output pair and corresponding prompt. In this case, we input the degradation task prompt $\mathcal{P}_{degr}(code_{fast})$ into $\mathcal{M}_{base}$ and minimize the IFT loss $\mathcal{L}_{\text{ift}}(\mathcal{P}_{\text{degr}}(code_{fast}), code_{slow})$.

3.2 Bidirectional Tree Editing

We propose bidirectional tree editing, which includes optimization-tree editing and degradation-tree editing, to explore the most efficient and least efficient code versions corresponding to a given code snippet, respectively. For example, for source code with a time complexity of $\mathcal{O}(n \log n)$, the optimization-tree editing yields a faster $\mathcal{O}(n)$ implementation, while the degradation-tree editing results in a slower $\mathcal{O}(n^2)$ implementation. This process emulates how developers modify code: increasing tree depth reflects iterative code modifications, while expanding tree width generates diverse code versions. Each node in the tree represents a distinct solution to the same programming problem, with child nodes corresponding to improved or degraded versions of their parent node's solution.

Optimization-Tree Editing. Starting with the source code src from an online programming platform as the root node, $\mathcal{M}_{opti}$ generates m optimized code versions corresponding to the root node, which serve as child nodes. Each child node is subsequently treated as a new root for further optimization, with each node having a degree of m. This process continues until the optimization tree reaches a predetermined height of h.

During the optimization tree generation, we apply pruning to avoid incorrect or inefficient optimizations. Specifically, if a child node's code is incorrect or its speedup relative to the parent node is less than 10%, further optimization of that node is terminated.

Degradation-Tree Editing. The degradation tree shares a structure similar to the optimization tree, with each child node representing a degraded version of its parent node's code. In this case, $\mathcal{M}_{degr}$ is used to generate less efficient code versions for the parent node. Similar to optimization-tree editing, we prune nodes that contain errors or have a

slowdown ratio less than 10% compared to their parent node. Appendix A offers an algorithmic description of our bidirectional tree editing process.

3.3 Progressive Code Acceleration Learning

We obtain multi-level CA code pairs from the constructed bidirectional tree. Using this data, we adapt the LLMs to the CA task through progressive knowledge acquisition and consolidation.

Multi-Level Data Construction. Given the optimization tree and the degradation tree with source code as the root node, we first derive three versions of the code: the least efficient code from the degradation tree (denoted as $slow$), the source code (denoted as src), and the most efficient code from the optimization tree (denoted as $fast$). Using these three code versions, we construct IFT data of CA tasks at three difficulty levels: (1) Easy: ($\mathcal{P}_{\text{opti}}(slow)$, src), *i.e.,* optimization from $slow$ to src; (2) Medium: ($\mathcal{P}_{\text{opti}}(src)$, $fast$), *i.e.,* optimization from src to $fast$; (3) Hard: ($\mathcal{P}_{\text{opti}}(slow)$, $fast$), *i.e.,* direct optimization from $slow$ to $fast$.

Progressive Knowledge Acquisition and Consolidation (PKAC). Inspired by curriculum learning (Bengio et al., 2009), we propose PKAC, which enables LLMs to gradually acquire more advanced CA knowledge through a three-stage training process of increasing difficulty. Additionally, to prevent forgetting previously acquired knowledge, each stage incorporates re-training on a subset of data from previous stages. All three stages utilize the IFT training strategy, yet with differentiated training data. Specifically, in the first stage, all easy-level data is used; the second stage includes the top 50% of the easy-level data with the highest speedup, along with all medium-level data; and the third stage incorporates the top 50% of the medium-level data with the highest speedup, along with all hard-level data.

4 A New CA Benchmark and Metric

4.1 Benchmark Construction

Existing CA benchmarks, while capable of evaluating LLMs' CA performance, exhibit several limitations:

(1) **Limited Language Coverage.** Current benchmarks (Shypula et al., 2024; Waghjale et al., 2024) typically focus on optimizing single programming languages like C++ or Python, neglecting the support for more programming languages.

(2) **Data Leakage Risks in Source Code.** The construction of current CA benchmarks often rely on traditional datasets, such as the CodeNet dataset (Puri et al., 2021) used in benchmarks like PIE (Shypula et al., 2024) and ECCO (Waghjale et al., 2024), which includes data only up to 2020 (Puri et al., 2021). If the LLM has access to this data during pre-training, it will lead to inaccurate evaluation (Xu et al., 2024).

(3) **Oversimplified Code Input.** Algorithmic efficiency gains are typically more pronounced in complex scenarios. For example, $\mathcal{O}(n^2)$ and $\mathcal{O}(n)$ implementations usually show negligible runtime differences when n is 1. This issue is overlooked in existing benchmarks, resulting in unstable runtime measurements where random time fluctuations may obscure actual optimization gains (Shypula et al., 2024).

(4) **Lack of Expert-Optimized References.** A significant deficiency in current CA benchmarks is their omission of reference points for achievable optimization ceilings. For example, reducing the time complexity of list summation from $\mathcal{O}(n)$ to $\mathcal{O}(1)$ is theoretically infeasible. Without references to expert-optimized code, there is a risk of pursuing unrealistic optimization targets.

To address these issues, we present a novel benchmark with four key enhancements:

(1) **Multi-lingual Support.** This benchmark covers 5 languages: C++, C, Go, Java, and Python.

(2) **Up-to-date Code Collection.** Source code is drawn from the latest programming problems on Codeforces, specifically those published after May 2024. For each problem, up to two accepted code submissions are selected.

(3) **Efficiency-Sensitive Code Inputs.** To underscore differences in efficiency, ten graduate students in computer science annotated 200 corresponding inputs for each code snippet, with each input requiring a minimum execution time of 5 ms on a single-core CPU.

(4) **Expert-Optimized Fast Code.** To establish realistic and achievable optimization targets, fifty algorithm competition experts provide at least 3 optimized versions for each code snippet, with the most efficient implementation selected as optimal target.

The specific statistics are shown in Table 1.

4.2 Metric Design

Traditional evaluation metrics (Shypula et al., 2024) for the CA task include: (1) Percent Optimized

	# Lang.	# Code	Code Date	# Input	Avg. Run Time	Expert Optimization
ECCO[EMNLP2024]	1	794 × 1	Pre-2020	100	76ms	✗
Mercury[NeurIPS2024]	1	256 × 1	Pre-2020	8	64ms	✗
PIE[ICLR2024]	1	978 × 1	Pre-2020	100	22ms	✗
BITE (Ours)	**5**	**1000 × 5**	**Post-2024**	**200**	**220ms**	✓

Table 1: Comparison between existing CA evaluation benchmarks and our benchmark. Our benchmark surpasses these benchmarks in terms of broader programming language coverage (*Lang*), larger-scale and up-to-date codebase (*Code* and *Code Date*), comprehensive time-sensitive code inputs (*Input* and *Avg. Run Time*), and expert-annotated optimized implementations (*Expert Optimization*).

(OPT). The proportion of successful code accelerations. (2) Speedup (SP). The acceleration ratio of the optimized code. (3) Percent Correct (COR). The proportion of generated code that is functionally correct, regardless of its time efficiency.

Although these metrics are widely used, they do not account for the optimal efficiency, which may encourage models to pursue unrealistic optimizations. Therefore, we introduce a new metric **Acceleration Progress**, denoted as **AP**, to measure the gap between the generated code and the optimal code, as follows:

$$\mathrm{AP}(old, new, opti) = \frac{\max(0, old - new)}{old - opti}, \quad (2)$$

where old, new, and $opti$ represent the execution times of the original code, the model-generated code, and the optimal code, respectively. $\mathrm{AP} \in [0, 1]$, with a value closer to 1 indicating that the new code is nearing optimal performance. If the model generates code with functional errors, we set new equal to old.

5 Experiments

5.1 Experimental Settings

Implementation Details. In the bidirectional tree editing, the height h is 5 and the node degree m is 10. During model training, we combine training data from C++, C, Go, Java, and Python, enabling a single model to handle all the investigated languages. During evaluation, we employ the code evaluation platform developed by Waghjale et al. (2024) to accurately assess the runtime of the generated code. The prompts used by BITE are presented in Appendix B. More detailed implementation details can be found in Appendix C.

Baselines. Our baselines are categorized into prompt-based and training-based methods. The prompt-based baselines include direct instructions for LLMs to perform the CA task, denoted as "Direct" (Shypula et al., 2024), a RAG-based strategy (Shypula et al., 2024), CodeRefine (Waghjale et al., 2024), and SBLLM (Gao et al., 2024). The training-based baselines comprise PIE-IFT, PIE-PerfCond (Shypula et al., 2024), ECCO-Execution, ECCO-Trajectory (Waghjale et al., 2024), Mercury-DPO (Du et al., 2024c), and Supersonic (Chen et al., 2024).

Benchmark and Metrics. Our experiments utilize our benchmark and three existing ones: ECCO (Waghjale et al., 2024), Mercury (Du et al., 2024c), and PIE (Shypula et al., 2024). In addition to our designed AP, we also follow the standard practice (Shypula et al., 2024) and use three other metrics: OPT, SP, and COR, which are detailed in Section 4.2.

5.2 Main Results

To verify the broad applicability of BITE, our experiments involve various types and sizes of LLMs, including $\text{Qwen}_{1.5-3B}$ (Qwen et al., 2025) and StarCoder_{3B} (Lozhkov et al., 2024). Evaluation results based on our benchmark and existing ones are shown in Table 2 and Table 3, respectively.

Evaluation on Our BITE Benchmark. Table 2 demonstrates the significant potential of BITE. A compelling piece of evidence is that BITE enables $\text{Qwen}_{1.5B}$ to outperform prompt-enhanced GPT-4 and training-based Qwen_{3B} across most programming languages and metrics. Using larger LLMs like Qwen_{3B} and StarCoder_{3B} further amplifies these improvements. Comparing different types of LLMs, we find that Qwen_{3B} demonstrates better average performance than StarCoder_{3B}, though StarCoder_{3B} exhibits a more pronounced advantage in Java across all metrics, highlighting the language-specific strengths of different LLMs.

Evaluation on Existing Benchmarks. Table 3 shows that BITE consistently delivers impressive

Method	LLM	C++				C				Go				Java				Python			
		AP	OPT	SP	COR	AP	OPT	SP	COR	AP	OPT	SP	COR	AP	OPT	SP	COR	AP	OPT	SP	COR
(I) Prompt Learning-Based Methods																					
Direct[ICLR2024]	GPT-4	10.2	8.2	27.4	35.6	11.6	6.7	105.6	38.5	9.6	3.2	104.3	17.4	13.4	12.0	103.8	28.7	12.4	5.0	101.3	34.5
RAG[ICLR2024]		10.6	12.3	151.3	24.8	13.5	8.1	125.7	32.8	12.5	6.7	136.8	13.5	14.3	18.7	111.2	20.5	13.8	8.1	108.8	27.6
CodeRefine[EMNLP2024]		33.2	24.7	174.3	56.8	35.2	16.4	213.6	54.6	39.7	18.7	294.6	37.4	43.4	34.2	154.8	56.4	43.4	14.5	134.4	47.5
SBLLM[ICSE2025]		41.4	32.1	204.2	47.8	44.6	19.7	225.8	48.9	35.7	20.0	287.3	31.9	47.5	49.8	174.2	54.3	45.2	17.3	152.3	43.5
(II) Training-based Methods																					
PIE-IFT[ICLR2024]	$Qwen_{3B}$	31.3	20.3	156.5	44.5	33.2	13.4	123.6	46.3	26.5	13.5	133.4	34.5	33.4	20.3	126.7	44.8	28.7	13.5	127.3	38.5
PIE-PerfCond[ICLR2024]		34.7	28.9	167.5	35.7	36.8	17.3	187.6	40.2	35.4	17.5	196.7	29.7	40.2	26.5	144.3	40.3	36.5	15.4	145.4	33.5
ECCO-Execution[EMNLP2024]		44.3	33.7	182.4	46.8	46.5	30.8	189.7	48.2	42.3	18.7	213.5	39.4	48.5	35.4	153.7	51.6	49.9	19.5	164.3	43.2
ECCO-Trajectory[EMNLP2024]		32.7	21.6	154.3	64.2	34.6	14.7	125.7	65.3	28.6	14.3	142.7	42.1	36.5	22.5	131.8	65.4	32.4	15.1	134.6	48.7
Mercury-DPO[NeurIPS2024]		45.3	34.5	193.2	45.7	47.9	23.3	219.8	47.8	44.7	25.7	294.3	36.7	52.3	53.2	167.7	60.5	49.7	18.7	166.7	40.3
Supersonic[IEEE TSE 2024]		43.2	31.1	177.8	50.6	43.4	20.5	222.5	54.6	-	-	-	-	-	-	-	-	-	-	-	-
BITE (Ours)	$Qwen_{1.5B}$	59.7	37.8	219.4	59.1	60.4	26.2	265.5	60.8	56.2	29.3	350.0	44.3	58.2	66.4	198.1	68.4	51.2	22.4	169.7	45.8
	$Qwen_{3B}$	**69.4**	**42.3**	**226.5**	**66.5**	**67.3**	**32.8**	314.3	**70.0**	**61.5**	**34.5**	**466.9**	**48.9**	60.7	68.2	198.5	69.2	**57.5**	**32.6**	173.8	**52.0**
	$StarCoder_{3B}$	64.5	41.2	224.5	62.7	65.7	30.0	**336.0**	69.2	58.7	29.9	375.5	46.0	**63.5**	**71.2**	**200.6**	**72.4**	55.2	24.2	**179.9**	46.2

Table 2: Model performance comparison on our BITE benchmark. The bold values represent the best results, while the underlined values indicate the second-best.

Method	LLM	PIE (C++)			ECCO (Python)			Mercury (Python)		
		OPT	SP	COR	OPT	SP	COR	OPT	SP	COR
(I) Prompt Learning-Based Methods										
Direct[ICLR2024]	GPT-4	8.5	115.3	80.2	12.4	132.5	43.5	13.4	125.2	74.1
RAG[ICLR2024]		43.2	232.6	66.4	21.7	154.3	35.4	18.2	126.5	53.1
CodeRefine[EMNLP2024]		54.3	289.4	**85.3**	22.4	185.7	61.2	24.3	144.3	**81.1**
SBLLM[ICSE2025]		52.1	293.4	72.3	26.5	176.4	54.2	32.0	176.8	76.5
(II) Training-based Methods										
PIE-IFT[ICLR2024]	$Qwen_{3B}$	34.6	223.7	43.5	21.1	165.4	47.1	23.4	113.4	51.2
PIE-PerfCond[ICLR2024]		37.4	273.5	40.3	32.4	178.5	46.3	25.6	122.3	44.3
ECCO-Execution[EMNLP2024]		38.7	254.5	51.2	27.8	177.6	54.3	31.4	144.2	55.4
ECCO-Trajectory[EMNLP2024]		31.5	204.2	66.4	23.4	143.2	64.2	25.4	121.8	73.2
Mercury-DPO[NeurIPS2024]		45.4	275.3	47.6	33.6	197.0	49.7	30.2	142.6	54.0
Supersonic[IEEE TSE 2024]		37.6	264.4	63.7	-	-	-	-	-	-
BITE (Ours)	$Qwen_{1.5B}$	70.1	319.5	75.7	37.6	220.7	58.7	43.2	214.8	72.1
	$Qwen_{3B}$	**77.9**	**370.6**	81.4	**41.5**	**257.4**	66.5	**44.3**	**236.8**	76.5
	$StarCoder_{3B}$	73.4	356.5	79.0	39.7	235.7	**67.8**	44.1	226.5	77.6

Table 3: Model performance comparison on traditional benchmarks. As shown in Table 1, these benchmarks lack ground-truth optimized implementations and therefore do not support the calculation of the AP metric.

performance across most metrics and benchmarks, reaffirming the strengths of our approach. However, for the COR metric, which focuses on code correctness, our models slightly lag behind CodeRefine-enhanced GPT-4 on the PIE and Mercury benchmarks. The PIE benchmark sources its code from the traditional CodeNet dataset (Shypula et al., 2024), and the Mercury benchmark consists of classic problems from LeetCode (Du et al., 2024c). It is possible that GPT-4 had already encountered the code from these two benchmarks during its pre-training phase, making it less prone to errors when optimizing such code. This also underscores the necessity of building an up-to-date CA benchmark.

5.3 Analysis of Bidirectional Tree Editing

Our bidirectional tree explores improved and degraded versions of source code by increasing the tree height h, while enhancing code diversity by expanding the node degree m. Figure 3, based on our benchmark, illustrates the average performance of BITE across five programming languages under varying values of h and m, from the perspectives of both optimization and degradation trees.

Optimization Tree Editing. As shown in (a.1) and (a.2) of Figure 3, increasing h and m significantly enhances the model's performance. An important observation is that when m is small, the benefit of increasing h is limited. For instance, when $m = 1$, increasing h from 1 to 5 results in an AP gain of only $36.0 - 34.6 = 1.4$ and an OPT gain of just $23.1 - 21.8 = 1.3$. However, when $m = 10$, the same increase in h yields an AP improvement of $57.1 - 48.8 = 8.3$ and an OPT improvement of $36.4 - 33.9 = 2.5$. This pattern aligns with theoretical expectations, as a richer set of code variants

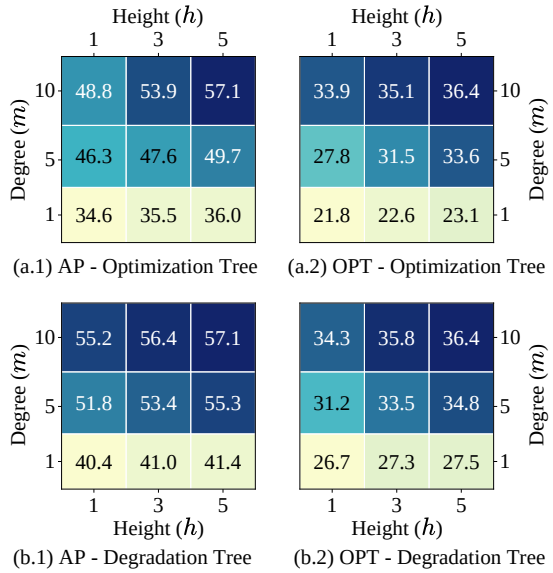

Figure 3: Qwen$_{1.5B}$-BITE performance across varying tree heights h and node degrees m, illustrating average results across five languages on our benchmark.

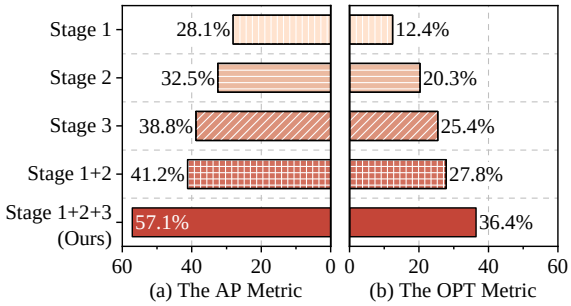

Figure 4: Impact of different stages in progressive code acceleration learning on Qwen$_{1.5B}$-BITE performance, evaluated on our benchmark.

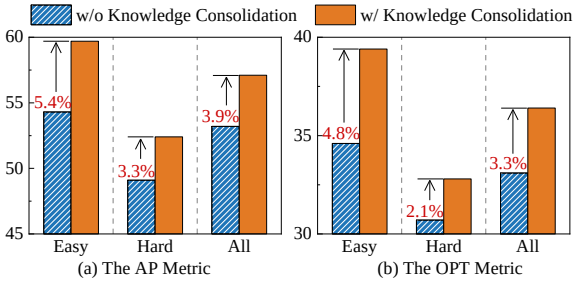

Figure 5: Ablation study examining the effects of knowledge consolidation in our progressive code acceleration learning, based on Qwen$_{1.5B}$-BITE and our benchmark.

allows for more thorough exploration of optimal code versions.

Degradation Tree Editing. The (b.1) and (b.2) of Figure 3 illustrate the positive effects of increasing the parameters h and m in the degradation tree. This suggests that training data derived from less efficient code variants can enhance model performance on CA tasks more effectively than source code from online programming platforms. However, we find limited benefits when increasing h from 3 to 5 while holding m constant. This may result from the introduction of numerous shallow degradation patterns, such as redundant computations, as the degradation tree explores increasingly suboptimal code states. Training on such simplistic data may limit the model's capacity for deeper reasoning about the CA task.

5.4 Analysis of Progressive CA Learning

The CA learning process in BITE employs the PKAC strategy based on multi-level training data. PKAC enables the model to tackle challenging code through progressive knowledge acquisition, while continuously reinforcing foundational knowledge through consolidation. We analyze this strategy from the perspectives of progressive knowledge acquisition and knowledge consolidation.

Progressive Knowledge Acquisition. Figure 4 illustrates the impact of different training stages on the model. From the results, we draw two key observations: (1) The gains from training increase progressively through stages 1, 2, and 3, likely due to the increasing difficulty of training data in each stage and a significantly higher acceleration ratio in stage 3. (2) Continuous multi-stage training yields further benefits. This indicates that initial learning on simpler CA data aids LLMs in understanding complex CA patterns, which aligns with human learning processes (Sweller, 1988).

Knowledge Consolidation. We perform ablation studies to underscore the importance of consistently consolidating foundational knowledge in multi-stage learning. In this experiment, we classify the source code in our benchmark into "Easy" and "Hard" categories based on cyclomatic complexity (McCabe, 1976) and evaluate the performance of various BITE variants on both categories. As illustrated in Figure 5, our knowledge consolidation strategy yields significant improvements for both code types, with particularly pronounced gains on simpler data. This finding suggests that focusing exclusively on complex data may cause the model to forget foundational CA knowledge.

Method	Similarity	Speedup
PIE-User[ICLR2024]	55.3	158.6
PIE-Problem[ICLR2024]	20.6	261.5
BITE (Ours)	**61.4**	**306.3**

Table 4: Comparison of similarity and speedup ratios between slow and fast code across different training datasets—code pairs from the same users (PIE-User) and the same programming problems (PIE-Problem) sourced from the online platform.

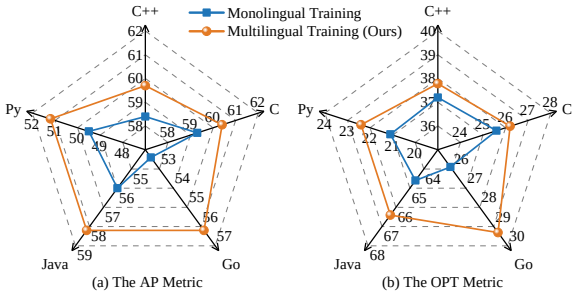

Figure 6: Comparison of Qwen$_{1.5B}$-BITE under monolingual training and multilingual training.

5.5 Discussion

In this subsection, we comprehensively discuss our approach around training data and methods.

Training Data. We compare the similarity (measured by CodeBLEU (Ren et al., 2020)) and speedup ratios of slow-fast code pairs generated by different training data construction methods. Besides our approach, we also consider the approach used by PIE (Shypula et al., 2024), which sources slow-fast code pairs from online platforms. PIE's code pairs are further categorized into two subtypes (Ye et al., 2024): (1) PIE-User (pairs from the same user's submissions) and (2) PIE-Problem (pairs from solutions to the same programming problem). As shown in Table 4, PIE-User pairs exhibit higher code similarity but limited speedup ratios, whereas PIE-Problem pairs demonstrate the opposite trend. In contrast, our training data achieves the best efficiency with the highest code similarity. Learning from such similar code pairs enables the model to precisely capture CA patterns.

Multilingual Training Strategies. BITE involves training across five programming languages, allowing for two strategies: (1) monolingual training, where separate models are trained for each language, and (2) multilingual training, where a single model is trained on data from all languages. We evaluate these strategies, and the results are presented in Figure 6. The results reveal that multilingual training consistently outperforms monolingual training across all languages. We attribute this success to the fundamental universality of CA patterns across programming languages, despite their syntactic differences. This finding suggests that multilingual training facilitates cross-lingual knowledge transfer, allowing the model to leverage CA patterns learned from one language to enhance performance in others.

6 Related Work

Prompt Learning-Based CA Methods. The prompt learning-based strategy enables training-free acceleration by giving CA-related prompts to LLMs. Shypula et al. (2024) shows that while LLMs can perform basic CA tasks with simple prompts, the performance, even with advanced models like GPT-4, remains suboptimal. Traditional prompt optimization methods, such as CoT (Wei et al., 2022), typically improve code acceleration but at the cost of code correctness reduction (Waghjale et al., 2024). To address this challenge, a CodeRefine strategy has been proposed (Waghjale et al., 2024; Peng et al., 2024), which uses important information generated by compilers, such as error reports, runtime, and execution results, to guide LLMs in further code correction or optimization. Additionally, Gao et al. (2024) emphasizes the significance of acceleration patterns, extracting rich code acceleration patterns from CA examples and using CoT to help models generalize these patterns to source code.

Training-Based CA Methods. Due to the scarcity of high-quality CA training data, researchers have adopted weak supervision approaches, primarily drawing training data from online programming platforms (Shypula et al., 2024). Blot and Petke (2025) finds that fine-tuning LLMs with CA data from these platforms leads to significant performance improvements. Recognizing that optimized code often preserves portions of the original code and that LLMs struggle with generating long outputs (Peng et al., 2023), Chen et al. (2024) proposed fine-tuning LLMs to output only the differences between the original and optimized code. Furthermore, preference-based learning strategies (Kaufmann et al., 2024), such as DPO (Rafailov et al., 2024), yield stable improvements (Du et al., 2024c). Although weakly-

supervised training data has proven effective, challenges like inconsistencies in coding style between slow and fast code, coupled with limited speedup, result in suboptimal training outcomes (Waghjale et al., 2024; Blot and Petke, 2025).

7 Conclusion

In this study, we propose BITE, a novel bidirectional tree-based progressive code acceleration learning paradigm. This approach comprises two key innovations: first, a bidirectional tree editing mechanism that systematically explores both optimized and degraded code variants; second, multi-level training data that enables progressive knowledge acquisition and consolidation through multi-stage training. Additionally, we develop a comprehensive CA evaluation benchmark and introduce a new metric that addresses the limitations of current CA evaluations. Our experimental results demonstrate that our approach significantly improves the performance of LLMs on CA tasks.

Acknowledgements

This work was supported by National Science Foundation of China (62476070, 62125201, U24B20174), Shenzhen Science and Technology Program (JCYJ20241202123503005, GXWD 20231128103232001, ZDSYS20230626091203008, KQTD20240729102154006) and Department of Science and Technology of Guangdong (2024A1515011540).

Limitations

While our BITE can effectively enhance LLMs' code acceleration capabilities, there are several limitations worth discussing. First, our work focuses exclusively on runtime optimization without considering memory consumption. This single-objective optimization may lead to solutions that achieve speedup at the cost of increased memory usage. Future research should explore multi-objective optimization strategies that balance both runtime efficiency and memory utilization. Second, although our bidirectional code editor could benefit from more powerful LLMs like GPT-4, the high computational cost of these models currently forces us to rely on fine-tuning open-source LLMs. Third, our study primarily focuses on conventional algorithmic code, leaving out other types such as neural network code. We plan to address this limitation by incorporating a broader range of training data in future work.

Ethical Considerations

Code acceleration is a well-studied task. The data and related resources used in our work are open source and widely used in existing research. To the best of our knowledge, our work fully complies with the ACL Ethics Policy.

References

Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. 2009. Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning (ICML)*, pages 41–48.

Aymeric Blot and Justyna Petke. 2025. A comprehensive survey of benchmarks for improvement of software's non-functional properties. *ACM Computing Surveys*.

Asma Charfi, Chokri Mraidha, Sébastien Gérard, François Terrier, and Pierre Boulet. 2010. Does code generation promote or prevent optimizations? pages 75–79.

Zimin Chen, Sen Fang, and Martin Monperrus. 2024. Supersonic: Learning to generate source code optimizations in c/c++. *IEEE Transactions on Software Engineering*.

Guodong Du, Runhua Jiang, Senqiao Yang, Haoyang Li, Wei Chen, Keren Li, Sim Kuan Goh, and Ho-Kin Tang. 2024a. Impacts of darwinian evolution on pre-trained deep neural networks. In *2024 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*.

Guodong Du, Junlin Lee, Jing Li, Runhua Jiang, Yifei Guo, Shuyang Yu, Hanting Liu, Sim Kuan Goh, Ho-Kin Tang, Daojing He, and Min Zhang. 2024b. Parameter competition balancing for model merging. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems (NeurIPS)*.

Mingzhe Du, Anh Tuan Luu, Bin Ji, Qian Liu, and See-Kiong Ng. 2024c. Mercury: A code efficiency benchmark for code large language models. In *Proceedings of the Thirty-eight Conference on Neural Information Processing Systems Datasets and Benchmarks Track (NeurIPS)*.

Shuzheng Gao, Cuiyun Gao, Wenchao Gu, and Michael Lyu. 2024. Search-based llms for code optimization. In *Proceedings of the 2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 254–266.

Spandan Garg, Roshanak Zilouchian Moghaddam, Colin B. Clement, Neel Sundaresan, and Chen Wu. 2022. Deepdev-perf: a deep learning-based approach

for improving software performance. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 948–958.

Michael Günther, Jackmin Ong, Isabelle Mohr, Alaeddine Abdessalem, Tanguy Abel, Mohammad Kalim Akram, Susana Guzman, Georgios Mastrapas, Saba Sturua, Bo Wang, Maximilian Werk, Nan Wang, and Han Xiao. 2024. Jina embeddings 2: 8192-token general-purpose text embeddings for long documents. *arXiv preprint arXiv:2310.19923*.

ISO/IEC25010. 2011. Systems and software engineering – systems and software quality requirements and evaluation (square).

Timo Kaufmann, Paul Weng, Viktor Bengs, and Eyke Hüllermeier. 2024. A survey of reinforcement learning from human feedback. *arXiv preprint arXiv:2312.14925*.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, Zhuang Li, Wen-Ding Li, Megan Risdal, Jia Li, Jian Zhu, Terry Yue Zhuo, Evgenii Zheltonozhskii, Nii Osae Osae Dade, Wenhao Yu, Lucas Krauß, Naman Jain, Yixuan Su, Xuanli He, Manan Dey, Edoardo Abati, Yekun Chai, Niklas Muennighoff, Xiangru Tang, Muhtasham Oblokulov, Christopher Akiki, Marc Marone, Chenghao Mou, Mayank Mishra, Alex Gu, Binyuan Hui, Tri Dao, Armel Zebaze, Olivier Dehaene, Nicolas Patry, Canwen Xu, Julian McAuley, Han Hu, Torsten Scholak, Sebastien Paquet, Jennifer Robinson, Carolyn Jane Anderson, Nicolas Chapados, Mostofa Patwary, Nima Tajbakhsh, Yacine Jernite, Carlos Muñoz Ferrandis, Lingming Zhang, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2024. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173*.

Daniel J Mankowitz, Andrea Michi, Anton Zhernov, Marco Gelmi, Marco Selvi, Cosmin Paduraru, Edouard Leurent, Shariq Iqbal, Jean-Baptiste Lespiau, Alex Ahern, et al. 2023. Faster sorting algorithms discovered using deep reinforcement learning. *Nature*, pages 257–263.

Thomas J McCabe. 1976. A complexity measure. *IEEE Transactions on software Engineering*.

Adrian Nistor, Tian Jiang, and Lin Tan. 2013. Discovering, reporting, and fixing performance bugs. In *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR)*, pages 237–246.

OpenAI. 2023. Gpt-4 technical report. *Preprint*, arXiv:2303.08774.

Ali Ouni, Marouane Kessentini, Mel Ó Cinnéide, Houari Sahraoui, Kalyanmoy Deb, and Katsuro Inoue. 2017. More: A multi-objective refactoring recommendation approach to introducing design patterns and fixing code smells. *Journal of Software: Evolution and Process*, page e1843.

Bowen Peng, Jeffrey Quesnelle, Honglu Fan, and Enrico Shippole. 2023. Yarn: Efficient context window extension of large language models. *arXiv preprint arXiv:2309.00071*.

Yun Peng, Akhilesh Deepak Gotmare, Michael Lyu, Caiming Xiong, Silvio Savarese, and Doyen Sahoo. 2024. Perfcodegen: Improving performance of llm generated code with execution feedback. *arXiv preprint arXiv:2412.03578*.

Ruchir Puri, David S Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, et al. 2021. Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks. *arXiv preprint arXiv:2105.12655*.

Qwen, :, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. 2025. Qwen2.5 technical report. *arXiv preprint arXiv:2412.15115*.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Stefano Ermon, Christopher D. Manning, and Chelsea Finn. 2024. Direct preference optimization: Your language model is secretly a reward model. *arXiv preprint arXiv:2305.18290*.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*.

Alexander G Shypula, Aman Madaan, Yimeng Zeng, Uri Alon, Jacob R. Gardner, Yiming Yang, Milad Hashemi, Graham Neubig, Parthasarathy Ranganathan, Osbert Bastani, and Amir Yazdanbakhsh. 2024. Learning performance-improving code edits. In *The Twelfth International Conference on Learning Representations (ICLR)*.

John Sweller. 1988. Cognitive load during problem solving: Effects on learning. *Cognitive science*, pages 257–285.

Siddhant Waghjale, Vishruth Veerendranath, Zhiruo Wang, and Daniel Fried. 2024. ECCO: Can we improve model-generated code efficiency without sacrificing functional correctness? In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 15362–15376.

Feng Wang, Zesheng Shi, Bo Wang, Nan Wang, and Han Xiao. 2025. Readerlm-v2: Small language model for HTML to markdown and JSON. *CoRR*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. 35:24824–24837.

Ruijie Xu, Zengzhi Wang, Run-Ze Fan, and Pengfei Liu. 2024. Benchmarking benchmark leakage in large language models. *arXiv preprint arXiv:2404.18824*.

Tong Ye, Tengfei Ma, Lingfei Wu, Xuhong Zhang, Shouling Ji, and Wenhai Wang. 2024. Iterative or innovative? a problem-oriented perspective for code optimization. *arXiv preprint arXiv:2406.11935*.

Shengyu Zhang, Linfeng Dong, Xiaoya Li, Sen Zhang, Xiaofei Sun, Shuhe Wang, Jiwei Li, Runyi Hu, Tianwei Zhang, Fei Wu, and Guoyin Wang. 2024. Instruction tuning for large language models: A survey. *arXiv preprint arXiv:2308.10792*.

Lei Zhao, Junlin Li, Lianli Gao, Yunbo Rao, Jingkuan Song, and Heng Tao Shen. 2022. Heterogeneous knowledge network for visual dialog. *IEEE Transactions on Circuits and Systems for Video Technology*, 33(2):861–871.

A Algorithm Description for Bidirectional Tree Editing

Algorithm 1 and 2 detail the optimization-tree editing and degradation-tree editing processes, respectively. We construct bidirectional trees using the breadth-first search algorithm. The two algorithms are identical except for their code editors and pruning strategies.

B Prompt Settings

Prompt $\mathcal{P}_{\text{opti}}$ for the Code Optimizer and BITE.

Below is a {LANGUAGE} program. Optimize the program and provide a more efficient version.
{LANGUAGE} Code:
{CODE}
Optimized Version:

Prompt $\mathcal{P}_{\text{degr}}$ for the Code Degrader.

Below is a {LANGUAGE} program. Degenerate the program and provide a less efficient versions.
{LANGUAGE} Code:
{CODE}
Degraded Version:

C More Implementation Details

Our training encompasses five programming languages: C, C++, Go, Java, and Python. During the bidirectional code editors training phase, we collected approximately 5,000 slow-fast code pairs per language from the CodeNet dataset (Puri et al., 2021) as training data. The CodeNet provides extensive code submissions from online programming platforms AIZU and AtCoder, along with basic code inputs, offering essential data support for our work. During the bidirectional tree editing, we gathered 10,000 code snippets per language from CodeNet to build bidirectional trees with a height $h = 5$ and node degree $m = 10$. We set the inference temperature of bidirectional editors to 1.0 to promote code diversity. The training of the bidirectional code editor and all stages of our progressive code acceleration learning utilize the same hyperparameters: 2 epochs with a learning rate of 1e-5. All experiments are carried out on a NVIDIA 8×A800-SXM4-80G machine.

D Baseline Details

Our baselines include both prompt-based and training-based methods. The prompt-based methods involve:

- **Direct** (Shypula et al., 2024): LLMs are prompted with source code and concise task descriptions to perform CA. Owing to the powerful instruction-following abilities of LLMs, direct prompt learning has proven highly effective across a broad range of tasks (Zhao et al., 2022; Wang et al., 2025; Du et al., 2024b,a).
- **RAG** (Shypula et al., 2024): Optimization examples similar to the input source code are incorporated into prompts for LLMs to reference. We use the Jina (Günther et al., 2024) retriever to select relevant CA data from the training set used by our code optimizer $\mathcal{M}_{opti}$.
- **CodeRefine** (Waghjale et al., 2024): This method leverages a compiler to provide quality feedback on LLM-generated code, guiding further code optimization.
- **SBLLM** (Gao et al., 2024): This multi-stage prompting strategy first employs an execution-based data selection method to choose the top-k best responses from multiple optimized code snippets generated by LLMs. These responses are then summarized into acceleration patterns, which are used in conjunction with a CoT strategy to generate better responses.

The training-based methods involve:

- **PIE-IFT** (Shypula et al., 2024): As described in Section 2, this method uses instruction data consisting of slow and fast code to fine-tune the LLM.
- **PIE-PerfCond** (Shypula et al., 2024): Building on PIE-IFT, this method incorporates the speedup ratio of slow-fast code pairs during training, enabling the model to more accurately capture the runtime efficiency of code.
- **ECCO-Execution** (Waghjale et al., 2024): This strategy incorporates code execution information, such as runtime and output, into the IFT data, to help the model accurately identify efficiency bottlenecks in the code.
- **ECCO-Trajectory** (Waghjale et al., 2024): This strategy provides multiple optimization trajectories of the code during model training, allowing the model to perform multi-stage optimization on the source code during inference, rather than generating efficient code in a single step.

Algorithm 1 Optimization-Tree Editing

Input: The source code src, the code input x, the code optimizer $\mathcal{M}_{opti}$, the maximum height of the tree h and the maximum degree of the node m.

Output: The optimization tree T_{opti}.

$queue, T_{opti} = [src], [src]$

▷ The maximum height of the optimization tree is h.

while $queue \neq []$ ***and*** $len(T_{opti}) < h$ **do**
  $current\_level = []$
  $queue\_len = len(queue)$
  for i ***in*** $\{1, \ldots, queue\_len\}$ **do**
    $parent = queue.pop()$
    ▷ Generate m child nodes based on $\mathcal{M}_{opti}$.
    $\{child_1, \ldots, child_m\} = \mathcal{M}_{opti}(parent)$
    for $child$ ***in*** $\{child_1, \ldots, child_m\}$ **do**
      ▷ Prune invalid or non-improving nodes
      if *Exec*$(parent, x)$ == *Exec*$(child, x)$ ***and*** $\frac{Time(parent,x)}{Time(child,x)} > 1.1$ **then**
        $queue.push(child)$
        $current\_level.append(child)$
      end
    end
  end
  $T_{opti}.append(current\_level)$
end
return T_{opti}

Algorithm 2 Degradation-Tree Editing

Input: The source code src, the code input x, the code degrader $\mathcal{M}_{degr}$, the maximum height of the tree h and the maximum degree of the node m.

Output: The degradation tree T_{degr}.

$queue, T_{degr} = [src], [src]$

▷ The maximum height of the degradation tree is h.

while $queue \neq []$ ***and*** $len(T_{degr}) < h$ **do**
  $current\_level = []$
  $queue\_len = len(queue)$
  for i ***in*** $\{1, \ldots, queue\_len\}$ **do**
    $parent = queue.pop()$
    ▷ Generate m child nodes based on $\mathcal{M}_{degr}$.
    $\{child_1, \ldots, child_m\} = \mathcal{M}_{degr}(parent)$
    for $child$ ***in*** $\{child_1, \ldots, child_m\}$ **do**
      ▷ Prune invalid or non-degrading nodes
      if *Exec*$(parent, x)$ == *Exec*$(child, x)$ ***and*** $\frac{Time(child,x)}{Time(parent,x)} > 1.1$ **then**
        $queue.push(child)$
        $current\_level.append(child)$
      end
    end
  end
  $T_{degr}.append(current\_level)$
end
return T_{degr}

- **Mercury-DPO** (Du et al., 2024c): In addition to collecting slow-fast code pairs, this method selects sub-optimal acceleration code from online platforms to construct preference data, which is then used for LLM training via the preference learning algorithm DPO.
- **Supersonic** (Chen et al., 2024): This strategy uses IFT to help LLMs output only the differences between fast and slow code and omit unchanged code sections, thus avoiding performance degradation caused by long outputs.