Performance-Aligned LLMs for Generating Fast HPC Code

Daniel Nichols*, Pranav Polasam†, Harshitha Menon*, Aniruddha Marathe*, Todd Gamblin*, Abhinav Bhatele†
*Lawrence Livermore National Laboratory, Livermore, CA, USA
†Department of Computer Science, University of Maryland, College Park, MD, USA

***Abstract*—Optimizing scientific software is a difficult task because codebases are often large and complex, and performance can depend upon several factors including the algorithm, its implementation, and hardware among others. Causes of poor performance can originate from disparate sources and be difficult to diagnose. Recent years have seen a multitude of work that use large language models (LLMs) to assist in software development tasks. However, these tools are trained to model the distribution of code as text, and are not specifically designed to understand performance aspects of code. In this work, we introduce a reinforcement learning based methodology to align the outputs of code LLMs with performance. This allows us to build upon the current code modeling capabilities of LLMs and extend them to generate better performing code. We demonstrate that our fine-tuned model improves the expected speedup of generated code over base models for a set of benchmark tasks from 0.9 to 1.6 for serial code and 1.9 to 4.5 for OpenMP parallel code.**

***Index Terms*—Large Language Models, Code Generation, Performance Optimization, Reinforcement Learning**

I. Introduction

DEVELOPING fast and scalable code is a difficult, but often necessary task for scientific software developers. It can require expert knowledge of the application domain, algorithm design, programming languages, and hardware. This is a challenging task for even serial code, and even more complex for parallel code. Further, programmers and performance engineers are often tasked with optimizing existing code, often not written by them, which requires understanding an existing codebase and the performance implications of changes. Large language models (LLMs) have emerged as a powerful tool for assisting in the software development process for a variety of tasks such as code completion, bug detection, and code summarization [1]. Recently, they have also been used with limited success to generate parallel code [2]. Yet they struggle to understand performance aspects of code as they are not designed for this task. Code LLMs are trained on code as text, and as a result, are not well-suited to reason about complex performance issues. Additionally, the code they generate does not consider performance and could be slow, despite being correct. This has been demonstrated in existing works that show LLMs often generate inefficient parallel code [3].

Creating artificial intelligence models that can generate faster code has the potential to significantly improve the productivity of software developers. By using performance-aware code LLMs, developers can focus on design and correctness without worrying about the performance implications of using LLMs to generate code. Additionally, as LLM-based tools become more integrated with software development workflows, developers will become more and more reliant on the quality of their outputs. Improving the performance of LLM generated code while maintaining its correctness will improve the quality of the target software being developed. Further, code LLMs that can write fast code can remove the need for every scientific and parallel programmer to be a performance expert in addition to their existing domain expertise.

It is non-trivial to create code LLMs that can generate faster code. Creating performance-aware code LLMs will require fine-tuning of LLMs using performance data, one challenge is creating such datasets. LLMs typically require very large, general datasets for training tasks, and it is challenging to create such large datasets for performance data. Arbitrary code can have a wide range of performance characteristics, and depend on many factors such as input data, hardware, and software environment. Due to the complexity in collecting performance data for arbitrary code, performance datasets are often small and/or narrow in focus. Further, even with such a dataset in hand, an LLM needs to be carefully fine-tuned to *align* its generated outputs with more performant code. There are many potential pitfalls here, for instance, improving the performance of generated code at the cost of correctness. Additionally, fine-tuned LLMs can learn a distribution too disjoint from their initial code distribution they modeled and lose their ability to generalize.

In order to overcome the challenges associated with collecting large scale performance data, we propose a new approach that combines a structured, narrow performance dataset with a more general synthetic code dataset for fine-tuning. We also propose two novel fine-tuning methodologies: (1) reinforcement learning with performance feedback (RLPF), which is based on reinforcement learning with human feedback (RLHF) [4], and direct performance alignment (DPA), which is based on direct performance optimization (DPO) [5]. We use these two approaches and the new dataset to align an existing code LLM to generate faster code. These proposed fine-tuning methodologies use fast and slow code pairs to fine-tune the LLMs to generate samples more similar to the fast code and less similar to the slow code. The aligned model is then evaluated on two code generation benchmarks and one code optimization benchmark. We find that the aligned model is able to generate code with higher expected speedups than that of the original model, while maintaining correctness. *Our proposed methodology is able to fine-tune existing state-of-the-art code LLMs to generate faster code than they were*

previously capable of generating.

This work makes the following important contributions:

- A code performance dataset that combines narrow, structured performance data with broad synthetic data to help models learn performance properties, but maintain their ability to generalize.
- Two novel fine-tuning methodologies, reinforcement learning with performance feedback (RLPF) and direct performance alignment (DPA), for aligning code LLMs to generate faster code.
- A fine-tuned, performance-aligned LLM that generates faster code than traditional code LLMs.
- A detailed study of the performance and correctness of the code generated by performance-aligned LLMs including serial, OpenMP, and MPI code. Additionally, an ablation study motivating the use of synthetic data to fine-tune code LLMs for performance.

II. Background

In this section, we detail concepts and prior works that our methodologies build upon. We provide a background into LLMs and their use for code generation. We further provide an overview of reinforcement learning and the Proximal Policy Optimization algorithm and their use in RLHF and DPO.

A. Large Language Models for Code

LLMs have been shown to be effective tools for many code generation tasks [1]. These LLMs are typically Transformer models [6] fine-tuned on large code datasets [7], [8] to model the probability distribution of code text data. These models can then be used to generate code, fill in missing code snippets, complete code snippets, and more. Code is generated by showing them a sequence of code text (as tokens) and using the model to predict the next token in the sequence. Getting good text generation with this method is not always straightforward, so additional sampling techniques such as *temperature* and *top-p* are often used to improve the quality of the generated text [9]. These control the randomness of the sampling process, with *temperature* controlling the entropy of the distribution and *top-p* controlling the tokens considered for sampling.

B. Reinforcement Learning and Proximal Policy Optimization

Reinforcement learning (RL) is a popular machine learning training paradigm where an agent model learns to interact with an environment to maximize a reward signal. This learning is typically accomplished by the agent iteratively taking actions in the environment, observing the results, and updating its policy to maximize the reward. While RL techniques have been popular for a number of years, they have recently been applied to LLMs due to their success in aligning LLM outputs with human preferences [4].

Proximal Policy Optimization (PPO) [10] is a popular RL algorithm that has been used to successfully fine-tune LLMs. It is a state-of-the-art algorithm that has become widely used due to its efficiency and robustness across a number of different tasks. A key difference between PPO and other RL algorithms is its use of clipping to prevent unusually large updates to the policy. PPO clips the ratio of the new agent policy and the previous agent policy to a range of $[1-\epsilon, 1+\epsilon]$. This prevents large weight updates, which can lead to instability in the training process. The clipped policy updates are combined with a value loss function (the reward signal) and an entropy loss function (to encourage exploration) to train the agent. After running many iterations of the training process, the agent learns to make decisions that optimize the reward signal. In this paper, we will train an agent (an LLM) to generate code that is fast (higher reward for faster, correct code).

Reinforcement learning with human feedback (RLHF) [4] applies these ideas to LLMs by first training a reward model on human preference data and then using algorithms such as PPO to optimize the LLM against this learned reward while constraining it to stay close to a supervised reference model. More recently, Direct Preference Optimization (DPO) [5] has been proposed as an alternative that reparameterizes the KL-regularized RLHF objective into a purely supervised loss over preference pairs, removing the need for an explicit reward model and online RL optimization. In this paper, we build on these techniques but use performance feedback instead of human feedback, and we compare both RL- and DPO-style training for aligning LLMs toward generating faster code.

III. Overview of Methodology

Figure 1 presents an overview of our methodology for aligning code LLMs to generate faster code. We start by creating a dataset that can be used to fine-tune an LLM to generate code that is both correct and fast (Section IV). To accomplish this, we collect a large, structured code dataset with performance data and test cases to measure correctness. This structured dataset is, however, not representative of the entire distribution of code we want an LLM to optimize so we ameliorate its shortcomings by generating a *synthetic* code dataset that covers a wider distribution.

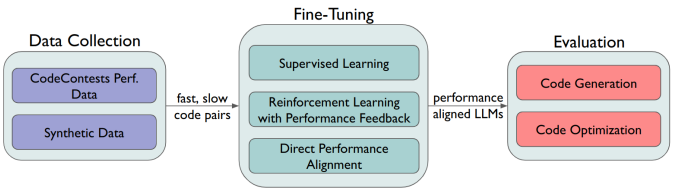

Fig. 1. An overview of the proposed methodology. We first collect a large dataset of fast and slow code pairs using coding contest submissions and synthetically generated data. Then we fine-tune three different LLMs on this data to generate faster code. Finally, we evaluate the fine-tuned models on code generation and optimization tasks.

These datasets are then used to *align* the outputs of an LLM with performance considerations. We employ three different techniques – supervised learning, reinforcement learning, and direct alignment, to fine-tune code LLMs (Section V). The models are aligned to answers that are not only correct, but also fast. Using the fine-tuned models we then generate code for a set of three different benchmark tasks for code generation and optimization (Section VI). These tasks measure

the correctness and performance of the generated code for coding problems outside the distribution of the training data.

IV. Data Collection and Labeling

In order to align LLMs to generate more performant output, we need to fine-tune them on performance data. Further, to apply the proposed fine-tuning methods, we need a dataset of code where we have a slow and a fast implementation of a particular problem. This type of structured performance data paired with source code is difficult to collect. It requires being able to build, execute, validate, and profile arbitrary code snippets, which is difficult to accomplish at scale. In this section, we describe our process of collecting a large performance dataset ($\mathcal{D}_c$). Additionally, we discuss how we extend the dataset with synthetic data ($\mathcal{D}_s$) to cover a wider distribution of code patterns. The final dataset $\mathcal{D}$ contains over 4.5 million code samples, distributed over three source languages (C++, Java, and Python) as shown in Table I.

TABLE I
NUMBER OF SAMPLES IN BOTH DATASETS BY SOURCE LANGUAGE.

Dataset ($\mathcal{D}$)	Runtime Data	C++	Java	Python	No. of Samples
CodeContests+Perf ($\mathcal{D}_c$)	✓	1.8M	0.9M	1.8M	**4.5M**
Synthetic ($\mathcal{D}_s$)	✗	5k	0	5k	**10k**

A. Performance Dataset Collection

We build our performance dataset using the CodeContests dataset introduced by DeepMind in [**?**]. This dataset contains coding contest problems and solutions from the Aizu [11], AtCoder [12], CodeChef [13], Codeforces [14], and HackerEarth [15] online competition platforms. In total there are 13,610 coding problems in the dataset. These range in difficulty from simple to very difficult, and cover a wide range of topics such as graph algorithms, dynamic programming, and search. Each problem in the dataset has a corresponding set of submissions from users, labeled as correct or incorrect. The number of submissions per problem ranges between tens and thousands. There are solutions in three different programming languages: C++, Java, and Python. Additionally, the dataset includes meta-data for the problem such as the problem statement, test cases, time limits, and memory limits.

This dataset is extremely valuable for our study as it provides a large amount of code samples along with the necessary tests to measure correctness and performance. More so, it contains many code samples that solve the same problem, but in different ways and with different runtimes. While many of the code contest websites record runtimes for submissions, the CodeContests dataset as provided by DeepMind does not include this information. We collect this data ourselves into a new dataset, *CodeContests-Perf* ($\mathcal{D}_c$), by executing each of the correct submissions and recording their runtimes. Each submission is run on all the test cases for its problem. Generally, there are between 5 and 20 test cases per problem. We create submission-runtime pairs using the average runtime over all the test cases. Each run is executed on a single core of an AMD EPYC 7763 CPU with a 2.45 GHz base frequency.

The final *CodeContests-Perf* dataset contains 4.5 million samples. The distribution of samples by source language is shown in Table I. There were a small fraction of submissions labeled as correct in the CodeContests dataset that errored or failed the test cases when we ran them. These are omitted from the final dataset. We also include code submissions that were marked as incorrect in the original dataset, however, we do not run them. These will eventually be useful to prevent the model from generating fast, but incorrect code.

B. Synthetic Data Generation

The amount of data and the availability of easy testing in the *CodeContests-Perf* dataset makes it a crucial component of our study. However, the distribution of code represented in the dataset is significantly different than that of the code that is typically found in production code. Coding contests generally award participants based on time-to-submission leading to users writing messy and/or disorganized code to solve problems as quickly as possible. Further, the types of problems typically found in coding contests such as depth-first search and dynamic programming, while an important subset, do not cover the full range of relevant problems that are found in production code, and in particular, in scientific computing.

To address the shortcomings of the *CodeContests-Perf* data, we generate an additional synthetic dataset $\mathcal{D}_s$ of fast and slow code samples. This is inspired by several recent works demonstrating the effectiveness of fine-tuning LLMs on synthetic data to improve performance on real tasks [8], [16], [17]. Gilardi et al. [16] even find that LLMs can outperform humans for many text annotation tasks. In our case of annotating code performance, real runtimes are the best annotation, but in the absence of runtime data, synthetic data is a promising candidate to obtaining labeled code performance data.

We use the Gemini-Pro-1.0 LLM model [18] to generate synthetic code samples as we found it to give the best outputs among a number of models we tested. We adapt the methodology in [8], where samples are generated using *seed* code snippets to get diverse outputs from the model. First, we create a dataset of 10,000 seed samples that are 1-15 line random substrings of random files from The Stack dataset [19], which is a large, 3TB dataset of permissively licensed code. Then the LLM is asked to generate three pieces of text: a problem statement inspired by the seed snippet, a fast solution to the problem, and a slow solution to the problem. This produces inherently noisy data, since the LLM does not always generate ideal (fast vs. slow) outputs. However, prior work has shown that gains in predictive performance from fine-tuning on synthetic data can outweigh the downsides of noisy data [8].

In total, we collect 10,000 synthetic samples, 5,000 in C++ and 5,000 in Python. While adding more synthetic samples would likely continue to improve the quality of the fine-tuned model, we found that limiting to 10,000 samples provided adequate model quality while operating within time/cost constraints of this study. Table I shows the distribution of samples.

Parallel Code Data Representation: As one of our key focuses is to improve models in generating and optimizing parallel HPC codes we ensure that many of the seed snippets used to generate synthetic data are parallel code. We filter The Stack [19] for CUDA, OpenMP, and MPI codes in c++, python, and Fortran. We also include a small percentage of Kokkos samples. In total 60% of the seeds are from parallel HPC sources. This ensures that there is representative parallel code in the synthetic dataset, which is crucial for improving a models' ability to generate fast parallel HPC code.

V. Aligning LLMs to Generate Faster Code: Proposed Fine-Tuning Approaches

LLMs have been shown to be capable of generating correct code with high frequency on several benchmarks [1], yet they do not always generate code that is efficient [3]. They require further fine-tuning to align them with performance considerations. In this section, we detail how we fine-tune LLMs with supervised learning and RL techniques to generate faster code. We utilize the dataset introduced in Section IV to train three different models using supervised learning, RL with performance feedback, and direct performance alignment.

A. Supervised Learning

In the first approach, we fine-tune a language model on the dataset of code from $\mathcal{D}$ to predict the next token in a sequence given previous tokens. For our methodology, we begin with a model that has already been trained on a large corpus of text and code, and then fine-tune it on a smaller dataset of coding problems and fast solutions.

We create two types of prompts using the samples in $\mathcal{D}$ to fine-tune the model. In the first type of prompt, we use a standard instruction prompt where the model is given a problem statement and a fast solution. Using the coding contest data in $\mathcal{D}_c$, we use the problem description as the instruction and randomly sample one of the five fastest solutions as the response. In the second type of prompt, we use a variation of the standard instruction prompt where the task is to optimize a given code snippet and the output is an optimized version of the code. For this, we use the problem description and one of the slowest 33% of solutions as the instruction, and one of the five fastest solutions as the response. Forming prompts from the synthetic dataset $\mathcal{D}_s$ is similar except we only have one slow and one fast solution for each problem, so we do not sample from ranges of solutions.

Over these prompts, the model is fine-tuned to minimize the cross-entropy loss between its predicted next token and the actual next token. We refer the reader to [20] for more details on fine-tuning LLMs for text generation. After fine-tuning, the model should have more fast code snippets in its training data and its probability distribution should shift toward faster code. Several prior works, however, have observed that methods more sophisticated than supervised fine-tuning are required to align LLM outputs with certain properties, such as safety and human preferences [4], [21].

Supervised Fine-Tuning Evaluation Metric: We evaluate the success of the supervised fine-tuning by measuring the perplexity of the tuned model over an evaluation dataset. Perplexity is inversely proportional to how confident a model is that a data sample is in the distribution it models. A lower perplexity is better and indicates the LLM is less "perplexed" by a particular sample. A model's perplexity over t tokens from a dataset $\mathcal{X}$ is given by Equation (1).

Predicted probability of token x_i given the previous tokens $x_{<i}$

$$\text{Perplexity}(\mathcal{X}) = \exp\left\{-\frac{1}{t}\sum_{i}^{t}\log\, p_\theta\left(x_i \mid x_{<i}\right)\right\} \quad (1)$$

B. Reinforcement Learning with Performance Feedback

To further align an LLM's outputs with performance considerations, we propose a new method, which we call reinforcement learning with performance feedback (RLPF). This method is inspired by the success of reinforcement learning with human feedback (RLHF) [4], which aligns LLM outputs with human preferences. RLHF uses human-labeled preference data to train a reward model that assigns rewards to LLM outputs that are more preferred by humans. This reward model is used in conjunction with reinforcement learning to fine-tune a LLM to generate outputs that are more preferred by humans. We adapt this method into RLPF that uses code performance instead of human feedback to fine-tune LLMs to generate faster code. Recent works have found success in using cleaner reward signals with RLHF or GRPO [**?**].

Reward Model: We first need to design a reward function that can be used to guide the reinforcement learning process. If we can automatically run, test, and measure the performance of a generated LLM output, then we can simply use a function of the recorded runtime as the reward. In our case, this is possible for the coding contests dataset $\mathcal{D}_c$, where we have unit tests available to run and test the generated code (see Section IV-A). This further highlights the utility of this dataset for our study.

As mentioned in Section IV-B, we want to be capable of generating fast code outside the context of coding contests i.e. we do not want to exclusively use the code contests data for RL fine-tuning. Since we may not be able to obtain runtime data for other arbitrary code samples, we need to train a reward model that rewards faster code more than slower code for samples where we cannot obtain runtime data. Fine-tuning LLMs for relative performance modeling was previously demonstrated by Nichols et al. [2] and, thus, a fine-tuned LLM is a viable candidate for the reward model.

To accomplish this we train a reward model (an LLM), r_θ, to predict a reward for a given code sample, where a higher reward indicates faster code. To train this model, we first use a subset of $\mathcal{D}$ to create a dataset of triplets (p, d_f, d_s) where p is a problem description and d_f and d_s are fast and slow code solutions to the problem, respectively. Using r_θ, we compute predicted rewards for d_f and d_s, and use these to calculate the loss function $\mathcal{L}_r$ in Equation (2).

predicted reward for fast code; predicted reward for slow code; adaptive margin; scales the reward based on runtimes

$$\mathcal{L}_r = -\log\left[\sigma\left(r_\theta\left(p, d_f\right) - r_\theta\left(p, d_s\right) - \mu\left(p, d_f, d_s\right)\right)\right] \quad (2)$$

This loss function is used to train r_θ to predict a higher reward for d_f than d_s. In Equation (2) σ, is the logistic function and μ is an adaptive margin as defined in Equation (3). The loss function in Equation (2) is adapted from Wang et al. [**?**] to include runtime information. It trains the reward model to generate rewards farther and farther apart for faster and slower code samples. As $r_\theta(p, d_f) - r_\theta(p, d_s)$ gets larger, the loss function tends towards zero. On the flip side, the loss increases as the difference between the rewards decreases or r_θ assigns a larger reward to the slower code. We utilize an adaptive margin μ to further scale the rewards based on how much faster the fast code is than the slow code:

$$\mu\left(p, d_f, d_s\right) = \begin{cases} \min\left\{\lambda, \frac{\text{runtime}(d_s)}{\text{runtime}(d_f)}\right\} & \text{if } p \in \mathcal{D}_c \\ 0 & \text{otherwise} \end{cases} \quad (3)$$

max margin value

speedup of d_f over d_s

Since we can train the reward model on both datasets $\mathcal{D}_c$ and $\mathcal{D}_s$, we can use the runtime information from $\mathcal{D}_c$ to scale the rewards appropriately. We use a max margin λ to prevent extremely large margins when d_s is very slow. Figure 2 provides an overview of the reward model fine-tuning process.

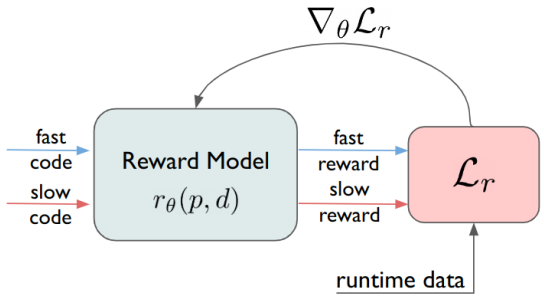

Fig. 2. An overview of the reward model fine-tuning process. The reward model outputs a reward for a fast and slow code sample. The loss function uses these rewards alongside runtime data to update the weights of the model so that its predicted rewards move farther apart for faster and slower code.

It is important to note that the reward model r_θ is not directly modeling code performance. Doing so would likely be impossible as performance can depend on a number of factors like hardware, input, etc. that are not accounted for in the input to the reward model. Instead, the reward model is trained to learn code structures and patterns that generally lead to better performance. This is another reason it is important to have a large dataset that covers a wide distribution of code, so that the model can learn these generalizations. Using the runtime data in $\mathcal{D}_c$ and the trained reward model, we can define a reward function $r(p, d)$ that assigns a reward to an LLM generated code sample. This reward function is defined in Equation (4).

$$r(p, d) = \begin{cases} -1 & \text{if } p \in \mathcal{D}_c,\ d \text{ incorrect} \\ \frac{\text{median\_runtime}(p)}{\text{runtime}(d)} - 1 & \text{if } p \in \mathcal{D}_c,\ d \text{ correct} \\ r_\theta(p, d) & \text{otherwise} \end{cases} \quad (4)$$

The model is penalized with a negative reward for incorrect code. If it generates correct code, then the reward is based on the speedup over the median runtime, $\text{median\_runtime}(d)$, from the submission already in the dataset. For the synthetic problems, we use the output of the reward model r_θ.

Reward Model Evaluation Metric: We evaluate the final reward model by its accuracy over an evaluation dataset. The accuracy is defined as the proportion of samples where the reward is larger for the fast code than for the slow code.

$$\text{acc}_{\text{reward}}(\mathcal{X}) = \frac{1}{|\mathcal{X}|} \sum_{(p, d_f, d_s) \in \mathcal{X}} \mathbb{1}\left[r_\theta(p, d_f) > r_\theta(p, d_s)\right] \quad (5)$$

Here $\mathbb{1}$ is the indicator function that returns 1 if the condition is true and 0 otherwise. A perfect accuracy of 1 indicates that the reward model always predicts a higher reward signal for the fast code sample than the slow code sample.

Reinforcement Learning: Using the reward function $r(p, d)$ and PPO [10], we align an LLM to generate faster code. We use the supervised fine-tuned model from Section V-A as the base model to fine-tune with RL as is common in RLHF [4]. We optimize the base model using the reward objective function in Equation (6).

$$\mathcal{L}_p = r(p, d) - \eta \text{KL}\left(\pi^{\text{RLPF}}(d \mid p) \parallel \pi^{\text{S}}(d \mid p)\right) \quad (6)$$

new model π^{RLPF} being fine-tuned with RL

supervised model π^{S}

Here KL is the Kullback-Leibler divergence and η is a hyperparameter that controls the divergence penalty. This penalty helps prevent the model from getting stuck in local optima or diverging too far from the original distribution of the supervised model.

During fine-tuning, a prompt is given to the base model (a coding problem or optimization task) and it generates a response. The reward function $r(p, d)$ is then used to compute a reward for the response either by running the generated code or getting a reward from the reward model. The reward is then used to compute the loss function $\mathcal{L}_p$ in Equation (6). The loss is then used to update the base model's parameters using PPO. The process is repeated for a number of iterations T or until the model converges. Figure 3 provides an overview of RLPF.

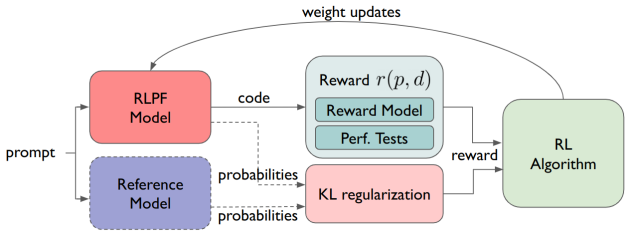

Fig. 3. The RLPF fine-tuning process. A prompt is given to the model and a reward is calculated based on the code it generates. Additionally, the KL-divergence between a reference model and the fine-tuned model is included in the reward to prevent deviating too far from the original distribution. Finally, PPO is used to update the model's parameters based on the reward.

RLPF Fine-Tuning Evaluation Metric: We measure the success of RLPF using two metrics: the mean reward and the magnitude of the KL-divergence over an evaluation dataset. The mean reward indicates how well the model is able to optimize the reward function. A higher mean reward is better and indicates that the model is generating faster code. The KL-divergence measures how far the fine-tuned model has diverged from the supervised model. The absolute magnitude of this is difficult to interpret, but it should be positive and low to indicate the fine-tuned model is not diverging too far.

Convergence: There are two optimization problems in RLPF: training the reward model r_θ and fine-tuning the PPO policy.

Reward model. The loss in Equation (2) is a Bradley–Terry style logistic loss on the scalar difference $\Delta r_\theta(p, d_f, d_s) = r_\theta(p, d_f) - r_\theta(p, d_s)$ with an additional margin $\mu(p, d_f, d_s)$ that is constant with respect to θ. This margin simply shifts the logit and increases the penalty on pairs where the runtime gap is large; it does not change the set of minimizers. As in standard pairwise preference learning, any r_θ that orders all fast/slow pairs correctly with a sufficient separation achieves a (near-)optimal value of $\mathcal{L}_r$ [**?**], [**?**]. In practice, because r_θ is a large neural network, the objective is non-convex, and, as is standard in LLM preference modeling, we rely on empirical convergence evidence to validate the reward model.

Policy optimization. The RL stage uses standard PPO with a KL penalty, as given in Equation (6). We do not modify the PPO update rule; RLPF only changes the scalar reward signal $r(p, d)$. Theoretical convergence properties therefore carry over unchanged under the usual assumptions of PPO [10].

C. Direct Performance Alignment

In recent work, Rafailov et al. [5] demonstrated an alternative approach that does not use reinforcement learning to align LLM outputs with certain properties. Their approach, called Direct Preference Optimization (DPO), uses a derivation of RLHF's reward objective (similar to Equation (6)) to directly update the model's parameters to align with a reward signal, rather than train a reward model and use RL. The derived loss takes a similar form to the reward loss in Equation (2). This DPO fine-tuning has many advantages over RLHF, such as requiring less computation, being easier to implement, and is generally more stable with less hyperparameters [5]. However, some works still find that RL fine-tuning can outperform DPO for certain tasks and datasets [**?**]. Thus, we adapt the DPO approach to compare it with RLPF. We propose Direct Performance Alignment (DPA), an adaptation of the training procedure and loss function from [5] that takes into account performance, to fine-tune an LLM to generate faster code. The proposed loss function in DPA is shown in Equation (7).

predicted probabilities from fine-tuned model π^P and supervised model π^S on fast (d_f) and slow (d_s) code

$$\mathcal{L}_d = -\log \sigma\Bigg(\beta \log \frac{\pi^P(d_f \mid p)}{\pi^S(d_f \mid p)} - \beta \log \frac{\pi^P(d_s \mid p)}{\pi^S(d_s \mid p)} - \mu(p, d_f, d_s)\Bigg) \quad (7)$$

Like with the reward loss in Equation (2), we utilize the adaptive margin μ from Equation (3) to scale the loss based on the runtime of the fast and slow code samples. This loss function can be used to fine-tune a base LLM to generate faster code without using RL. To compute the loss, we need fast and slow code pair model predictions from the model being fine-tuned and a base reference model (the supervised model). Then the loss from Equation (7) is used to update the weights of the model being fine-tuned. This process is iteratively repeated for a number of iterations T or until the model converges. This DPA fine-tuning process is portrayed in Figure 4.

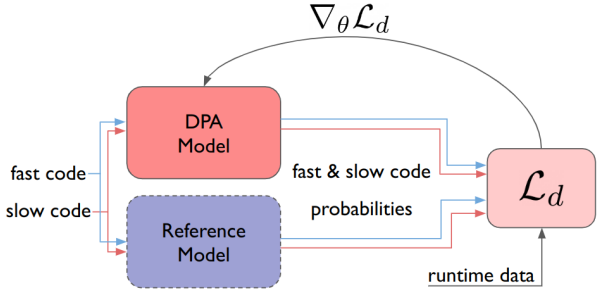

Fig. 4. The DPA fine-tuning process. The model being fine-tuned and a reference model are used to generate probabilities for a fast and slow code sample. These probabilities, combined with runtime data, are used to compute a loss and update the model's parameters.

DPA Fine-Tuning Evaluation Metric: The success of DPA fine-tuning can be measured using a similar accuracy metric to the reward model from RLPF. Since we do not have a direct reward signal like in Equation (2), we can instead measure how often the difference in log probabilities between the fine-tuned model and the supervised model for the fast code, i.e. $\log \frac{\pi^P(d_f|p)}{\pi^S(d_f|p)}$ is greater than the log probability difference for the slow code, i.e. $\log \frac{\pi^P(d_s|p)}{\pi^S(d_s|p)}$. This is shown in Equation (8).

$$\text{acc}_{\text{dpa}}(\mathcal{X}) = \frac{1}{|\mathcal{X}|} \sum_{(p,d_f,d_s)\in\mathcal{X}} \mathbb{1}\left[\frac{\pi^P(d_f \mid p)}{\pi^S(d_f \mid p)} > \frac{\pi^P(d_s \mid p)}{\pi^S(d_s \mid p)}\right] \quad (8)$$

Convergence: DPA differs from DPO only by subtracting an example-dependent margin $\mu(p, d_f, d_s)$ from the Bradley–Terry [**?**] logit in Equation (7). For each training triple (p, d_f, d_s), the DPO objective rewards the model when the log-likelihood ratio between the fine-tuned and supervised models is larger for the fast code than for the slow code, i.e.

$$\log \frac{\pi^P(d_f \mid p)}{\pi^S(d_f \mid p)} - \log \frac{\pi^P(d_s \mid p)}{\pi^S(d_s \mid p)} > 0.$$

Adding the margin simply tightens this condition to

$$\log \frac{\pi^P(d_f \mid p)}{\pi^S(d_f \mid p)} - \log \frac{\pi^P(d_s \mid p)}{\pi^S(d_s \mid p)} > \mu(p, d_f, d_s),$$

without changing the fixed points of the objective, since μ does not depend on the model parameters. As a result, the convergence analysis in [5], particularly Equations 6 and 7, applies directly to DPA. Adding margins to monotone loss functions is a common ML technique pre-dating LLMs [**?**].

VI. Evaluation Tasks

It is important to quantify how well the models do on downstream tasks after fine-tuning. In this section we present two different tasks, code generation and optimization, to evaluate how well the training methodologies in Section V improved the LLMs ability to generate fast code. We further detail an ablation study to motivate the use of synthetic data.

A. Code Generation

To evaluate the ability of the models to generate fast code, we utilize two sets of coding problems. The first is a subset of 100 coding contest problems from the CodeContests dataset [**?**] that were removed from the training set. We can provide the model with the problem statement and use it to write a solution to the problem. We can then run the code and measure both its *correctness* and *performance*. Correctness can easily be tested using the problem's provided unit tests.

In addition to the coding contest problems, we also evaluate the models on the ParEval benchmark [3], which is a collection of parallel code generation problems for evaluating the ability of LLMs to generate correct and efficient parallel code. We narrow our focus to a subset of 180 problems, namely the serial, OpenMP [22], and MPI [23] problems. We include OpenMP and MPI problems to evaluate the models' ability to generate fast parallel code. The problems in ParEval range a wide variety of domains, such as linear algebra, graph algorithms, sorting, etc. The problems are designed to be challenging and require the generation of efficient code. The ParEval benchmark provides a great way to test the LLMs on problems unlike what is in their training data (coding contests). This is the most important aspect of our benchmarking; we include results across coding contests for completeness, but demonstrating that the fine-tuned model can write fast complex, parallel scientific code on previously unseen problems is a key demonstration of our approaches' efficacy.

Further benefits of using ParEval come from the selection of problems it contains. ParEval is designed to test common scientific kernels, such as iterative solvers, linear algebra, graph algorithms, etc. They are intentionally designed to mimic real world HPC scientific kernels. Thus, if a model is able to generate fast code on these problems, or optimize them, then it is likely it will be useful for single kernel optimization in real-world applications. While optimizing HPC applications often involves more than just single kernel optimization, the ability to automatically optimize individual kernels is a crucial first step towards building out full application methods.

Code Generation Evaluation Metrics: We evaluate the generated code on two metrics: *correctness* and *performance*. To study correctness we adopt the popular pass@k metric from Chen et al [1]. This metric measures the probability that if an LLM is given k attempts to write a correct solution, it will succeed. Equation 9 shows how this value can be estimated using N generated samples from an LLM. Typically the average pass@k over a set of prompts is reported and, as LLMs have progressed, only the pass@1 value is reported. We refer the reader to [1] for further discussion of pass@k.

$$\text{pass@}k = \frac{1}{|P|}\sum_{p\in P}\left[1 - \binom{N - c_p}{k} \Big/ \binom{N}{k}\right] \quad (9)$$

Number of samples generated per prompt; Set of prompts; Number of correct samples for prompt p

To evaluate the performance of the generated code, we use the speedup$_n$@k metric introduced by Nichols et al [3]. This metric measures the expected max speedup over a baseline implementation if the LLM is given k attempts to write a solution. The speedup$_n$@k metric is defined in Equation 10. We refer the reader to [3] for a complete derivation of this metric. For the coding contest problems, we use the median submission runtime as the baseline. For the ParEval problems, we use the baselines provided by the benchmark.

$$\text{speedup}_n\text{@}k = \frac{1}{|P|}\sum_{p\in P}\sum_{j=1}^{N}\frac{\binom{j-1}{k-1}}{\binom{N}{k}}\frac{T_p^*}{T_{p,j,n}} \quad (10)$$

runtime of baseline for prompt p; runtime of sample j of prompt p on n processors

B. Code Optimization

In addition to generating code, we also evaluate the ability of the models to optimize existing code. This is accomplished by providing a code snippet and instructing the model to generate an optimized version of it. To evaluate this task we use the functions in the PolyBench benchmark suite [24]. This is comprised of 30 unique kernels that are typically used to test compiler optimizations and auto-tuning tools. We provide an existing kernel implementation to the LLM and instruct it to generate an optimized implementation. We can then evaluate the correctness and performance of the generated code.

Code Optimization Evaluation Metrics: We evaluate the generated code on the same metrics as the code generation task: *correctness* and *performance*. We use the same pass@k metric (Equation (9)) to evaluate correctness. To evaluate performance, we use speedup$_n$@k (Equation (10)), except with the baseline being the runtime of the original kernel.

C. Synthetic Data Ablation Study

Finally, we test our hypothesis that training on synthetic data helps the models' ability to generalize and prevents it from over-fitting to code contest data. To accomplish this we train the models exclusively on the code contests dataset $\mathcal{D}_c$ without any of the synthetic dataset $\mathcal{D}_s$. We then evaluate the models on the code generation (Section VI-A) and code optimization (Section VI-B) tasks. We compute the same pass@k and speedup$_n$@k metrics and compare the impact of the synthetic data on the models' performance. Of most interest is the performance on the ParEval and PolyBench benchmarks, as these are the most different from the training data.

VII. Experimental Setup

Using the large performance dataset $\mathcal{D}$ from Section IV and the training methodology introduced in Section V, we can

now fine-tune LLMs to generate faster code. Once fine-tuned, these models can then be evaluated on the benchmarks detailed in Section VI. This section details the base models for fine-tuning, the data subsets for each fine-tuning task, how we implement the fine-tuning process, and the experimental setup used to evaluate the fine-tuned models.

A. Base Model for Fine-Tuning

Each of the training methodologies introduced in Section V begins with a base LLM that has already been trained and fine-tunes it further. We select the Deepseek-Coder 6.7B model [7] as the base for the supervised fine-tuning (Section V-A). It is a 6.7B parameter code LLM released by Deepseek-AI that is trained on 2T tokens comprised of mostly code with a context length of 16k tokens. We select this model due to its good performance on code generation tasks [25] and due to other works finding it a better base model for fine-tuning than the popular CodeLlama models [8]. Furthermore, its 6.7B parameter size makes it tractable for end-users to use to generate code themselves on consumer hardware. While Deepseek-Coder is a strong base model for our studies, the proposed methodologies can be applied to any existing code LLM. To demonstrate this we also fine-tune Llama3.1-8B [**?**].

For the remaining two fine-tuning methods, RLPF and DPA, we use the supervised fine-tuned deepseek model as the base. This is in line with the methodologies in [4], [5] and ensures that the model being aligned is within the distribution of the text data it is trying to model. Additionally, we use Deepseek-Coder 6.7B as the base for the reward model. The final set of models used for comparison is shown in Table II.

TABLE II
MODELS USED FOR COMPARISON IN THIS PAPER.

Model Name	Description	Fine-Tuning Methodology
DS	Deepseek-Coder 6.7B base model	—
DS+SFT	DS after supervised fine-tuning	Section V-A
DS+RLPF	DS+SFT after RLPF fine-tuning	Section V-B
DS+DPA	DS+SFT after DPA fine-tuning	Section V-C
LL	Llama3.1-8B base model	—
LL+SFT	LL after supervised fine-tuning	Section V-A
LL+RLPF	LL+SFT after RLPF fine-tuning	Section V-B
LL+DPA	LL+SFT after DPA fine-tuning	Section V-C

B. Data Setup

We fine-tune the LLMs using the dataset $\mathcal{D}$ from Section IV. We set aside 100 contests from the CodeContests dataset for the code generation evaluation task. The dataset is further split into smaller datasets for each fine-tuning task. The supervised fine-tuning dataset, $\mathcal{D}_{\text{SFT}}$, is comprised of 40% of the full dataset, $\mathcal{D}$, and the remaining 60% is used for the reinforcement fine-tuning dataset, $\mathcal{D}_{\text{RLPF}}$, and the direct performance alignment dataset, $\mathcal{D}_{\text{DPA}}$. These two datasets can be the same since the alignment fine-tuning tasks are disjoint. The $\mathcal{D}_{\text{RLPF}}$ dataset is further split into 66% for the reward model dataset, $\mathcal{D}_{\text{REWARD}}$, and 33% for the reinforcement learning dataset, $\mathcal{D}_{\text{RL}}$. During each fine-tuning stage we set aside 5% of the respective dataset for evaluation (i.e. 5% of $\mathcal{D}_{\text{REWARD}}$ is set aside to calculate the reward model accuracy after training). All of the dataset splits are stratified so that the proportion of code contest to synthetic data is equal to the original dataset.

When creating prompt, fast code, and slow code triplets (p, d_f, d_s) from $\mathcal{D}_c$ for RLPF and DPA fine-tuning, we select d_f randomly from the top 5 fastest solutions. We then select d_s from the slowest 50% of the solutions. Additionally, a random 5% subset of slow solutions are replaced with an incorrect solution. This is to ensure that the model is not just learning to generate fast code, but also to avoid generating incorrect code. We use the fast-slow code pairs from $\mathcal{D}_s$ for the triplet.

C. Fine-Tuning Setup

In order to implement the fine-tuning we extend the TRL Python library [26]. TRL provides existing implementations of RLHF and DPO, which we modify to use our custom rewards, loss function, and datasets. We fine-tune the models on a single node with four 80GB A100 GPUs and two AMD EPYC 7763 CPUs. We detail our setup parameters here and point the reader to [**?**], [4] for further conversation on RL training setup nuances.

1) Supervised Fine-Tuning Hyperparameters: We fine-tune the supervised model for three epochs over the $\mathcal{D}_{\text{SFT}}$ dataset. We use bfloat16 precision and a global batch size of 64 (1 sample per GPU and 16 gradient accumulation steps). To fine-tune in parallel we make use of the PyTorch fully sharded data parallelism (FSDP) implementation [27], which shards model parameters across ranks to save memory. Furthermore, we use the Adam optimizer [28] and an initial learning rate of 1.41×10^{-5} (the default in TRL [26]).

2) Reward Model Fine-Tuning Hyperparameters: The reward model is fine-tuned with the same hyperparameters as the supervised model (Section VII-C1), except it is fine-tuned for only one epoch over the $\mathcal{D}_{\text{REWARD}}$ dataset. We use a max margin of $\lambda = 3$ for the margin function $\mu(p, d_f, d_s)$. Experimentally we found this max margin to limit the runtime ratio from getting too large.

3) RLPF Fine-Tuning Hyperparameters: We fine-tune the RLPF model for four PPO epochs over the $\mathcal{D}_{\text{RL}}$ dataset. We use a global batch size of four and a learning rate of 1.41×10^{-5}. The KL regularization coefficient is initialized to $\eta = 0.1$. When sampling outputs from the fine-tuned and reference model we use sampling with a top-k of 0 and a top-p of 1.0. All of these values are taken from best conventions in [26].

4) DPA Fine-Tuning Hyperparameters: The DPA model is fine-tuned for 1 epoch over the $\mathcal{D}_{\text{DPA}}$ dataset with a global batch size of four. We employ a learning rate of 1×10^{-7} in the AdamW optimizer [29] (taken from [5], [26]). We experimentally found $\beta = 0.6$ most stable for training.

D. Evaluation Setup

For the code generation tasks we use each of the LLMs to generate code for the prompts in the evaluation subset of $\mathcal{D}_c$ and ParEval. We generate 20 samples per prompt with a temperature of 0.2 and a top-p of 0.95 following standard practices LLM code benchmarks [3]. For the optimization task we similarly generate 20 optimized versions of each kernel in the PolyBench benchmark suite using each of the models.

The generated code is run on a single AMD EPYC 7763 CPU. For the ParEval OpenMP tests we report results on 8 cores and use 512 ranks for the MPI tests. We use the tests in CodeContests and ParEval to record correctness and runtime of the generated code. For the optimized PolyBench kernels we test correctness and runtime against the original implementations. All runtimes are averaged over five runs.

Model Selection: As our main contribution is the fine-tuning methodologies to align LLMs to generate faster code, our primary comparisons are between the LLMs with different fine-tuning methodologies applied to them. Each of these fine-tuned LLMs are compared with the performance of (A) human written baselines, (B) the base LLM, and (C) the other fine-tuned LLMs. We do not include large commercial LLMs since their weights are not open and, thus, we cannot apply our fine-tuning methodologies to them and evaluate its impact.

VIII. Results

With the fine-tuned models from Section V we now evaluate their code generation abilities on the tasks in Section VI. This section presents the results from fine-tuning and evaluation.

A. Fine-Tuning Results

Our training process ran on four 80GB A100 GPUs. The SFT fine-tuning took approximately 12 hours, while DPA took approximately 18 hours. RLPF took much longer with 6 hours for reward model training and 34 hours for RL fine-tuning. In general the DPA fine-tuning took less GPU memory as no reward model needed to be stored in memory and was much easier to fine-tune than RLPF. This is in line with similar benefits from using DPO over RLHF [5].

We record the fine-tuning metrics on the 5% evaluation datasets at the end of each fine-tuning step. The DS+SFT model yields an evaluation perplexity of 1.62. It is generally difficult to reason about specific perplexity values, but values near 1 show a strong ability to model the underlying text distribution. Since perplexity is the exponential of cross-entropy (see Equation (1)) a perplexity value of 1.62 means that the cross-entropy between predicted probabilities is ≈ 0.48.

The RLPF reward model achieves a final evaluation accuracy of 93% after one epoch of training calculated using Equation (5). This means that in 93% of samples the model assigns a higher reward signal to faster code than slower code. This is a strong result as the success of RL-based LLM fine-tuning is highly dependent on the quality of the reward model [**?**]. Using this reward model the DS+RLPF model is then able to achieve a mean reward of 1.8 and a KL divergence of 0.29. This means that DP-RLPF is getting a positive mean reward, while maintaining a similar distribution to the original model.

Finally, we see that the DS+DPA model achieves an evaluation accuracy of 87% calculated as shown in Equation (8). This is not as high as the RLPF reward model, but is still a strong result. The log-probability difference between DS+DPA and the reference model for fast code samples is greater than for slow code samples in 87% of the evaluation dataset.

B. Code Generation Results

Figures 5 and 6 show the correctness and performance results of each fine-tuned model on the code generation tasks. We see a promising trend in pass@1 scores in Figure 5 where the fine-tuned models improve in correctness over the baseline model. The RLPF models show the most improvement across all tasks. These improvements can be attributed to training over more data and, in the case of the RLPF and DPA models, using incorrect samples as negative rewards. Improving the correctness of the models is a strong result considering that the primary goal of this work is to improve the performance while keeping the correctness levels the same.

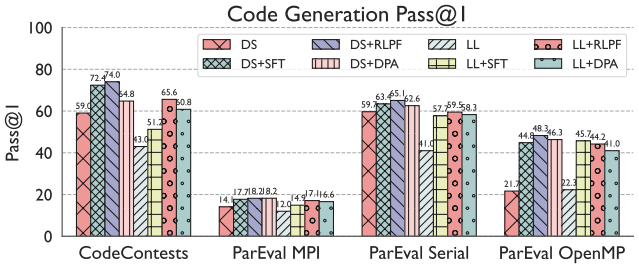

Fig. 5. Correctness results for each model on the code generation tasks. Each of the fine-tuned models shows an improvement in correctness over the baseline model with the RLPF models showing the most improvement.

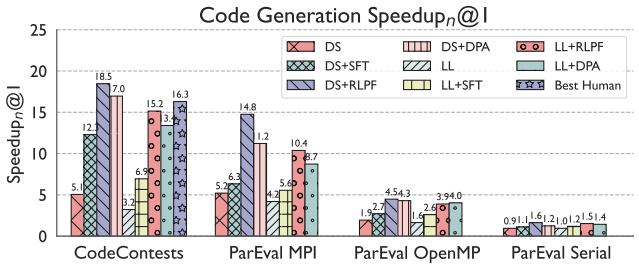

Fig. 6. Speedup results for each model on the code generation tasks. OpenMP runtimes are on 8 cores and MPI runtimes are on 512 ranks. The RLPF models are the best performing model across all benchmarks. "Best Human" shows the speedup of the best human coding contest submission over the median.

Figure 6 further details the speedup results for each fine-tuned model. We present the speedup results for OpenMP on 8 cores and MPI on 512 ranks with a sequential implementation as the baseline. Across all four benchmarks DS+RLPF produces faster code than the other models. In the case of the code contests and ParEval serial problems, the speedup$_1$@1 value is easy to interpret. For instance, in the case of the serial ParEval problems, DS+RLPF generates code with an expected max speedup of 1.6x over the sequential baseline. We see the same order of model performance across all the benchmarks with DS+RLPF performing the best, followed by DS+DPA, DS+SFT, and DS. The Llama models performed worse than the Deepseek models, but the same trends hold.

We analyzed a random subset of 50 outputs from each model to better understand how RLPF and DPA were able to generate faster code. For coding contest problems, they typically used better algorithms or data structures to improve performance and often frequently used compiler intrinsics like

vectorization or optimization hints. For OpenMP and MPI problems, they made better use of the parallelisms constructs. For example, standard DS frequently "handrolls" implementations of scan operations, while RLPF models would use the native MPI_Scan function for better performance.

Memory Usage Analysis: We recorded peak memory usage during the runtime of each generated code sample. We find little to no statistically significant difference in memory usage from the original codes and between the different models. This is due to most of the coding contest and ParEval problems we optimized were functions that were passed already allocated memory buffers. The optimizations did not need to change the memory usage of the code to achieve better performance.

Energy Usage Analysis: Due to the fact that most product HPC systems do not enable RAPL counters for users due to security concerns, we were not able to collect power usage data on the same machine as the code generation evaluation. However, we were able to collect power on a separate Intel Ultra 9 288V processor from a local machine to demonstrate power results. We note that runtime trends on this processor were similar to those presented later on the AMD EPYC CPU, confirming the validity of the power usage data. From the power usage data and runtimes we can estimate the energy usage of each generated code sample. We find the average energy usage across ParEval serial problems to be 1.35 mWh for DS, 1.36 mWh for DS+SFT, 1.29 mWh for DS+DPA, and 1.27 mWh for DS+RLPF. There is little difference in energy usage except for DPA and RLPF due to shorter runtimes.

Cross-Architecture Generalization: Table III shows the speedup results for RLPF models on two different CPU architectures: AMD EPYC 7763 and Intel Sapphire Rapids. We observe that the models generalize well across architectures achieving strong speedups on both systems. The speedup values are similar between architectures, demonstrating that the fine-tuned models learn performance optimizations that transfer across different hardware platforms.

TABLE III
SPEEDUP@1 RESULTS FOR RLPF MODELS ON AMD EPYC 7763 AND INTEL SAPPHIRE RAPIDS ARCHITECTURES.

Model	CodeContests	ParEval		
		MPI	OpenMP	Serial
AMD EPYC 7763				
DS+RLPF	18.5	14.8	4.5	1.6
LL+RLPF	15.2	10.4	3.9	1.5
Intel Sapphire Rapids				
DS+RLPF	17.2	12.7	4.9	1.3
LL+RLPF	12.0	8.6	4.4	1.3

C. Code Optimization Results

Figure 7 shows the correctness and performance results when using the fine-tuned models to optimize PolyBench kernels. DS is omitted because it is only a code completion model and was not trained to optimize code inputs. We first see that all three fine-tuned models transform the input code to a correct output code with relatively high accuracy. While provably correct compiler optimizations may seem more desirable, LLM optimizations can be applied at a higher level of abstraction and include natural language comments to explain the transformation to a developer.

```
/* Output from baseline DS model */
float dot(const float* y, const float* x, int n) {
  float acc = 0.0f;
  for (int i = 0; i < n; ++i) {
    acc += y[i] * x[i];
  }
  return acc;
}

/* Output from DS+RLPF model */
float dot(const float* __restrict y, const float* __restrict
↪  x, int n) {
  float acc = 0.0f;
  #pragma omp parallel for simd reduction(+:acc)
  for (int i = 0; i < n; ++i) {
    acc += y[i] * x[i];
  }
  return acc;
}
```

Listing 1: Example of outputs from the baseline DS and DS+RLPF models. The base DS model, when prompted for a dot product implementation, typically gives the canonical C/C++ dot implementation. The DS+RLPF model, on the other hand, outputs a much faster implementation by default.

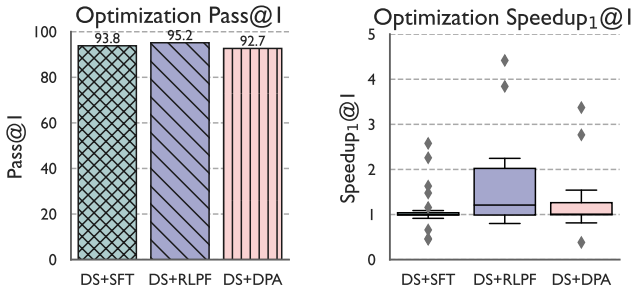

Fig. 7. pass@1 (left) and speedup$_1$@1 (right) results for optimizing PolyBench kernels. The distribution of speedup values over the 30 benchmarks is shown on the right. DS+RLPF has further outliers at 11.6 and 22.4.

We show the distribution of speedup$_1$@1 per PolyBench benchmark in Figure 7 rather than an average to highlight the spread of results. The speedup results show that DS+RLPF is the best performing model. It is able to produce an expected max speedup greater than 1 in 26 out of the 30 benchmarks. In the case of the 3mm kernel (three matrix multiplies) it is able to get up to 22.4x expected speedup. As in Section VIII-B, we sample 50 optimizations from each model to analyze by hand. Many of the optimizations come from loop unrolling and/or cache friendly data access patterns.

Listing 1 shows an example of the different outputs between a baseline model and the RLPF model. The baseline, while not incorrect, tends to produce a canonical C/C++ dot product implementation. The RLPF model, on the other hand, will default to generating a much faster implementation by using OpenMP parallelism and SIMD vectorization.

D. Synthetic Data Ablation Study Results

We further highlight the use of synthetic data in the fine-tuning process in Figures 8 and 9. Results for DS+RLPF are shown since it is the best performing model. We see a

general improvement in both correctness and performance of generated code when incorporating synthetic data versus fine-tuning on just coding contest data. Correctness improves for all benchmarks (Figure 8) and, notably, even improves on the coding contest benchmarks. The synthetic data is able to improve generalization even within the coding contest domain.

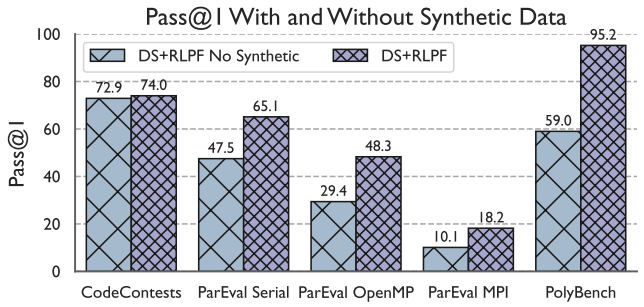

Fig. 8. pass@1 results for DS+RLPF on each task with and without synthetic data in the fine-tuning dataset. For all tasks, the model fine-tuned on synthetic data produces correct code at a higher rate.

The speedup results in Figure 9 show that fine-tuning with synthetic data also helps the models produce faster code. Only in the case of the coding contests and ParEval serial problems do we see a decrease or no change in speedup$_n$@1. However, these differences are small. The performance increases for OpenMP, MPI, and PolyBench are much more significant. incorporating synthetic performance data into the fine-tuning process has prevented the models from overfitting code contest data and enabled them to generalize better to new tasks.

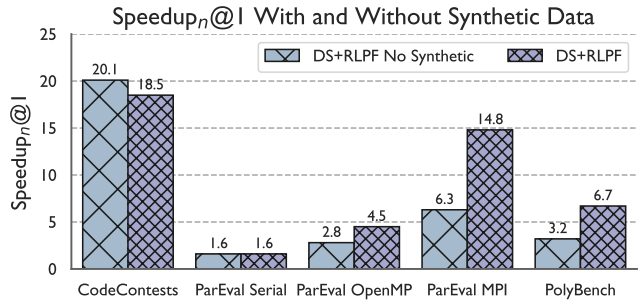

Fig. 9. speedup$_n$@1 results for DS+RLPF with and without synthetic data in the fine-tuning dataset. For OpenMP, MPI, and PolyBench tasks, the model fine-tuned on synthetic data produces faster code, while the coding contest and ParEval serial problems show a slight decrease or no change.

IX. Related Work

In this section we detail relevant works that study LLMs for HPC and parallel code or use RL to align LLMs for code.

LLMs, like OpenAI Codex [1] and DeepSeek [7] are revolutionizing how developers approach their coding tasks. These models are trained on vast datasets that include code repositories, documentation, high quality programming problems and solutions. They have shown incredible potential in many software-related tasks.

In HPC, researchers are particularly interested in using LLMs for generating parallel code [2], [3], [30]. Nichols et al. [2] proposed HPCCoder, a model fine-tuned on HPC data, to generate parallel code, label OpenMP pragmas, and predict performance. Chen et al. designed the LM4HPC [30] framework to facilitate the research and development of HPC software and proposed OMPGPT [31] for generating OpenMP pragmas. Despite their popularity, LLMs still struggle at generating efficient code [3]. Our work addresses this shortcoming.

While RLHF [**?**] has been critical for boosting the performance of LLMs by incorporating human feedback into the reward model [5], it is not specialized to code and furthermore does not consider performance. Another work by Mankowitz et al. [32] trains a deep RL agent, AlphaDev, to discover sorting algorithms that outperformed previously known human benchmarks. However, the training process is limited to a single algorithm at a time and does not yield a model that can generate fast code for general problems. To address this gap and enable LLMs to generate faster versions of general code, we introduced RLPF and DPA.

Further works have explored alternatives to RLHF for RL on LLMs. GRPO [**?**] uses a group reward across many reward signals and finds that it yields better convergence. These works are tangential to ours as you could theoretically substitute different RL techniques into our data and reward approach.

X. Limitations

The models are trained and evaluated on C++, Java, and Python, and the parallel experiments use OpenMP and MPI running on a CPU architecture, so the results may not directly transfer to other languages or accelerators (e.g., GPUs). However, we note that accelerating parallel CPU code for three of the most common programming languages is already a significant result. There are also threats to internal and construct validity. Correctness is assessed using available test suites, which may not fully cover the space of legal inputs. Furthermore, while we take care to evaluate our models on distributions outside of their training data, it is possible that some evaluation distributions may see lower performance improvements. We believe that the ParEval unit tests are sufficiently strong to catch correctness errors and the problems cover a wide range of scientific kernels with over 420 problems.

XI. Discussion

As presented in Section IX, there has been preliminary work into evaluating the performance of LLM generated code and constructing large agentic frameworks to design new algorithms and faster solutions. However, little work has actually explored *aligning model weights* with a more performance-oriented objective, such that the model can now yield faster code solutions by default. The proposed models in this work are tangential to works that explore large agentic solutions as the models yielded in this work can be high performing components within such systems. Furthermore, this work tries to address the more fundamental problem of single and/or few function optimization, which prior work [3] has shown that LLMs still struggle with. This is an essential problem to solve before LLMs are ready to handle full application optimization. Recent work [33] has shown that LLM software engineering

agents cannot even solve simple tasks in small HPC code repositories, much less for larger, full applications.

The importance of this work, thus, lies in its capability to align existing models to generate faster code. While fine-tuning techniques exist for model alignment they are not readily applicable to performance-oriented tasks. These techniques require large amounts of data to train, which is difficult to obtain for performance data without significantly sacrificing the generality of the data. Furthermore, HPC code is a low-resource domain, meaning that performance-oriented data is even more scarce. A main contribution of this work is the insight to combine large scale performance data with more general synthetic data to deal with scarcity. Furthermore, novel reward functions are introduced to handle the combination of real performance runtimes and latent reward signals from a model. Finally, while coding contest problems are included in the evaluation for completeness' sake, the crux of the evaluation is in the ParEval and PolyBench tasks, which are designed to contain realistic single and few function code tasks from the HPC and compiler domains.

XII. Conclusion

In this paper, we have explored fine-tuning LLMs to learn code structures and patterns that lead to better performance. To accomplish this, we first collected a large performance dataset from coding contests and extended it with synthetically generated samples to cover a wider code distribution. We then introduced two novel fine-tuning methodologies, RLPF and DPA, that align LLMs with faster code outputs. We have demonstrated that such techniques increase the expected performance of generated code over baseline LLMs while maintaining correctness for both serial and parallel codes. Our results demonstrate that we can fine-tune existing code LLMs to generate faster code than previously capable.

Acknowledgment

This work was performed under the auspices of the U.S. Department of Energy (DOE) by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-JRNL-2016980). This material is based upon work supported in part by the U.S. DOE, Office of Science, Office of Advanced Scientific Computing Research, through solicitation DE-FOA-0003264, "Advances in Artificial Intelligence for Science", under Award Number DE-SC0025598. This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. DOE Office of Science User Facility, operated under Contract No. DE-AC02-05CH11231 using NERSC award DDR-ERCAP0034262.

References

[1] M. Chen and et al, "Evaluating large language models trained on code," 2021.

[2] D. Nichols, A. Marathe, H. Menon, T. Gamblin, and A. Bhatele, "Hpc-coder: Modeling parallel programs using large language models," ser. ISC '24, may 2024.

[3] D. Nichols, J. H. Davis, Z. Xie, A. Rajaram, and A. Bhatele, "Can large language models write parallel code?" in *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '24. New York, NY, USA: Association for Computing Machinery, 2024.

[4] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. L. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, J. Schulman, J. Hilton, F. Kelton, L. Miller, M. Simens, A. Askell, P. Welinder, P. Christiano, J. Leike, and R. Lowe, "Training language models to follow instructions with human feedback," 2022.

[5] R. Rafailov, A. Sharma, E. Mitchell, S. Ermon, C. D. Manning, and C. Finn, "Direct preference optimization: Your language model is secretly a reward model," 2023.

[6] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," *CoRR*, vol. abs/1706.03762, 2017. [Online]. Available: http://arxiv.org/abs/1706.03762

[7] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. K. Li, F. Luo, Y. Xiong, and W. Liang, "Deepseek-coder: When the large language model meets programming – the rise of code intelligence," 2024.

[8] Y. Wei, Z. Wang, J. Liu, Y. Ding, and L. Zhang, "Magicoder: Source code is all you need," *arXiv preprint arXiv:2312.02120*, 2023.

[9] A. Holtzman, J. Buys, L. Du, M. Forbes, and Y. Choi, "The curious case of neural text degeneration," in *International Conference on Learning Representations*, 2020. [Online]. Available: https://openreview.net/forum?id=rygGQyrFvH

[10] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," 2017.

[11] "Aizu," https://judge.u-aizu.ac.jp/onlinejudge/.

[12] "Atcoder," https://atcoder.jp/.

[13] "Codechef," https://www.codechef.com/.

[14] "Codeforces," https://codeforces.com/.

[15] "Hackerearth," https://www.hackerearth.com/.

[16] F. Gilardi, M. Alizadeh, and M. Kubli, "Chatgpt outperforms crowd workers for text-annotation tasks," *Proceedings of the National Academy of Sciences*, vol. 120, no. 30, Jul. 2023. [Online]. Available: http://dx.doi.org/10.1073/pnas.2305016120

[17] L. Ben Allal, A. Lozhkov, G. Penedo, T. Wolf, and L. von Werra, "Cosmopedia," 2024. [Online]. Available: https://huggingface.co/datasets/HuggingFaceTB/cosmopedia

[18] G. Team, "Gemini: A family of highly capable multimodal models," 2023.

[19] D. Kocetkov, R. Li, L. Ben Allal, J. Li, C. Mou, C. Muñoz Ferrandis, Y. Jernite, M. Mitchell, S. Hughes, T. Wolf, D. Bahdanau, L. von Werra, and H. de Vries, "The stack: 3 tb of permissively licensed source code," *Preprint*, 2022.

[20] H. Naveed, A. U. Khan, S. Qiu, M. Saqib, S. Anwar, M. Usman, N. Akhtar, N. Barnes, and A. Mian, "A comprehensive overview of large language models," 2024.

[21] DeepSeek-AI, "Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning," 2025. [Online]. Available: https://arxiv.org/abs/2501.12948

[22] "OpenMP Application Program Interface. Version 4.0. July 2013," 2013.

[23] M. Snir, *MPI–the Complete Reference: The MPI core*, ser. MPI: The Complete Reference. Mass, 1998. [Online]. Available: https://books.google.com/books?id=x79puJ2YkroC

[24] J. C. S. Grauer-Gray, "Polybench," https://web.cs.ucla.edu/~pouchet/software/polybench/, 2012.

[25] "Big code models leaderboard - a hugging face space by bigcode," 2023. [Online]. Available: https://huggingface.co/spaces/bigcode/bigcode-models-leaderboard

[26] L. von Werra, Y. Belkada, L. Tunstall, E. Beeching, T. Thrush, N. Lambert, and S. Huang, "Trl: Transformer reinforcement learning," https://github.com/huggingface/trl, 2020.

[27] Y. Zhao and e. al, "Pytorch fsdp: Experiences on scaling fully sharded data parallel," *Proc. VLDB Endow.*, vol. 16, no. 12, p. 3848–3860, aug 2023.

[28] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2015. [Online]. Available: http://arxiv.org/abs/1412.6980

[29] I. Loshchilov and F. Hutter, "Fixing weight decay regularization in adam," *CoRR*, vol. abs/1711.05101, 2017. [Online]. Available: http://arxiv.org/abs/1711.05101

[30] L. Chen, P.-H. Lin, T. Vanderbruggen, C. Liao, M. Emani, and B. de Supinski, "Lm4hpc: Towards effective language model application in high-performance computing," in *OpenMP: Advanced Task-Based, Device and Compiler Programming*, S. McIntosh-Smith, M. Klemm, B. R. de Supinski, T. Deakin, and J. Klinkenberg, Eds. Cham: Springer Nature Switzerland, 2023, pp. 18–33.

[31] L. Chen, A. Bhattacharjee, N. Ahmed, N. Hasabnis, G. Oren, V. Vo, and A. Jannesari, "Ompgpt: A generative pre-trained transformer model for openmp," *arXiv preprint arXiv:2401.16445*, 2024.

[32] D. J. Mankowitz, A. Michi, A. Zhernov, M. Gelmi, M. Selvi, C. Paduraru, E. Leurent, S. Iqbal, J.-B. Lespiau, A. Ahern *et al.*, "Faster sorting algorithms discovered using deep reinforcement learning," *Nature*, vol. 618, no. 7964, pp. 257–263, 2023.

[33] J. H. Davis, D. Nichols, I. Khillan, and A. Bhatele, "Pareval-repo: A benchmark suite for evaluating llms with repository-level hpc translation tasks," 2025. [Online]. Available: https://arxiv.org/abs/2506.20938