Modeling Parallel Programs using Large Language Models

Daniel Nichols, Aniruddha Marathe, Harshitha Menon, Todd Gamblin, Abhinav Bhatele

Department of Computer Science, University of Maryland,
Lawrence Livermore National Laboratory

ABSTRACT

In the past year a large number of large language model (LLM) based tools for software development have been released. These tools have the capability to assist developers with many of the difficulties that arise from the ever-growing complexity in the software stack. As we enter the exascale era, with a diverse set of emerging hardware and programming paradigms, developing, optimizing, and maintaining parallel software is becoming burdensome for developers. While LLM-based coding tools have been instrumental in revolutionizing software development, mainstream models are not designed, trained, or tested on High Performance Computing (HPC) problems. In this abstract we present a LLM fine-tuned on HPC data and demonstrate its effectiveness in HPC code generation, OpenMP parallelization, and performance modeling.

1 INTRODUCTION

As we enter the exascale era the scale and complexity of scientific software grows at an unprecedented rate. Developing and managing software is becoming increasingly difficult for developers. Sophisticated development tools are needed to help developers write code, debug issues, and optimize performance. In the past year, large language models (LLMs) have been shown to be effective at many of these tasks. However, most LLMs are not designed for, trained on, or tested on HPC problems. This abstract primarily addresses this problem.

In our approach we first collect a large corpus of HPC code. Then, we fine-tune three LLMs, namely GPT-2 [5], GPT-Neo [2], and PolyCoder [6], on this data set. After comparing the performance of these models, we select the best performing model for further evaluation. This model is then used to generate HPC code, label for loops with OpenMP pragmas, and predict the relative performance of HPC routines.

2 DATA COLLECTION

In order to fine-tune a LLM to be able to perform HPC tasks we need a large corpus of HPC code. To obtain this we collect code from a large number of GitHub repositories. We select those that have greater ≥ 3 stars, C/C++/Fortran as a primary language, and an HPC related tag.

The source files from these repositories are then deduplicated and filtered. The deduplication is accomplished by computing the sha256 hash of each file and removing those that have the same hash. According to Allamanis [1], removing duplicate data when training LLMs can significantly improve performance. Finally, files over 1MB and under 15 tokens are filtered out in order to exclude large library headers and small metadata files. Before deduplication and filtering, the data set contains $\approx$ 2GB of data with only $\approx$ 1.6GB being left afterwards.

SC '23, November, 2023, Denver, CO
2023. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

3 FINE-TUNING

Using the HPC data set we fine-tune three language models: GPT-2, GPT-Neo, and PolyCoder. These are selected based on their pre-training data set and model size. GPT-2 is the smallest model with 1.5B parameters and was trained on a web crawl data set consisting of only natural language. GPT-Neo is a larger model with 2.7B parameters and was trained on a combination of web crawl data and code data. Finally, PolyCoder is the same size as GPT-Neo but was trained on a data set of only code.

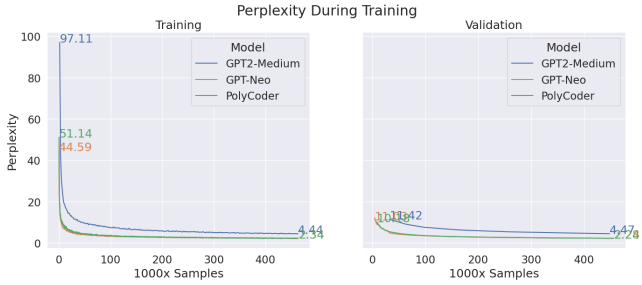

Figure 1: Perplexity during fine-tuning.

We fine-tune each of these models on the HPC data set for next token prediction where model learns to predict the next token in a sequence given the previous tokens. Figure 1 shows the perplexity of the models on the training and validation sets over the course of an epoch. The perplexity is a measure of how well the model is able to predict the next token in a sequence and is also the exponential of the loss (lower is better). All models train to a low perplexity demonstrating that they are able to model the HPC data set. GPT-Neo and PolyCoder score higher than GPT-2 likely due to their larger size and pre-training data.

4 TEXT GENERATION

Using the fine-tuned models we generate a set of HPC specific functions and evaluate the correctness and performance of the generated code. The HPC functions include sequential, OpenMP, and MPI based parallelism. For each of these functions we generate 1, 10, and 100 samples and compute the pass rate for each set of samples. Figure 2 shows the pass rate for each model and number of samples. Notably, native PolyCoder is bad at writing HPC and parallel code. By fine-tuning it on HPC data we are able to improve its performance significantly. Furthermore, PolyCoder+HPC, performs better than GPT-Neo+HPC and GPT-2+HPC.

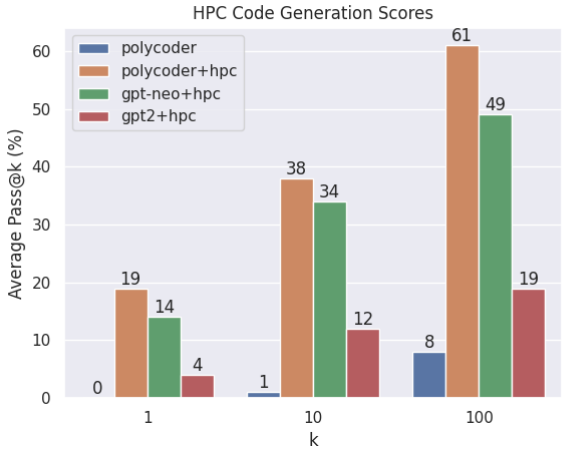

Figure 2: Fine-tuning models on HPC data significantly improves performance when generating HPC code.

We also evaluate the performance of the generated code by comparing it to an original implementation. Figure 3 shows the speedup of the generated code over the original implementation with PolyCoder+HPC. The speedup is always >1 indicating that the generated code is not just correct, but also performant.

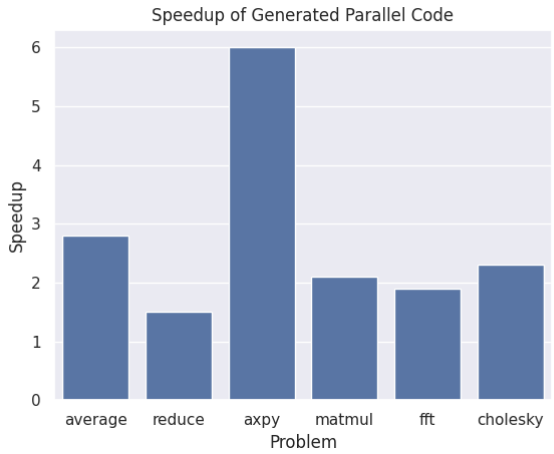

Figure 3: Speedup of generated code from PolyCoder+HPC over original implementations. The model not only generates correct code, but also performant code.

5 OPENMP PRAGMA LABELLING

In addition to text generation we also evaluate the ability of the models to label for loops with OpenMP pragmas. We accomplish this by first extracting all the for loops with OpenMP pragmas from the data set. We then fine-tune the models to take a for loop as input and generate the OpenMP pragma for that loop. The accuracy is computed by comparing the generated pragma to the original pragma in two ways: textual and functional equivalence. Figure 4 shows the accuracy of the models on both of these metrics. PolyCoder, when fine-tuned on HPC data, is able to label OpenMP pragmas with up to 97% accuracy, outperforming the native PolyCoder.

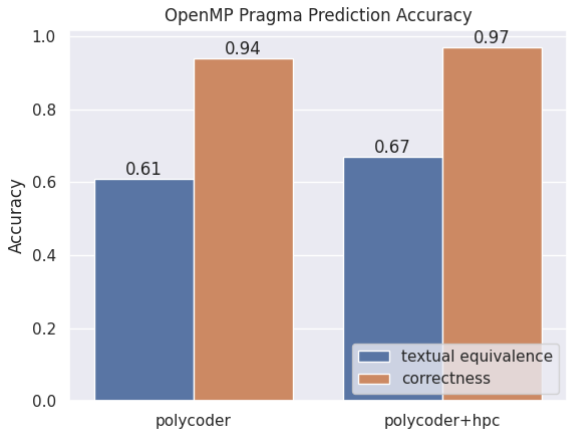

Figure 4: PolyCoder+HPC exhibits significantly higher accuracy than native PolyCoder for OpenMP labelling.

6 PERFORMANCE MODELING

We also test the model's ability to model performance attributes of HPC code by using it to predict relative performance across git commits. We accomplish this by first running all the commits of the Kripke [4] and Laghos [3] repositories for a total of 830 commits. The models are further fine-tuned to take the code before and after a commit as input and predict whether the code got faster (1) or stayed the same/slower (0). Figure 5 shows the accuracy of the models on this task. Both models perform well at predicting whether code got faster or slower across a git commit, however, PolyCoder+HPC performs slightly better with an accuracy of 88%.

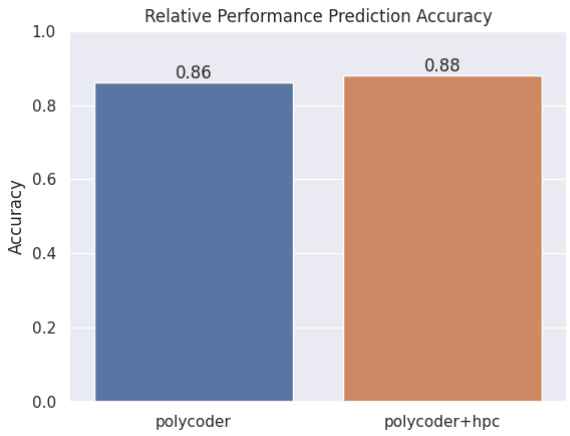

Figure 5: Both models perform well at predicting whether code got faster or slower across a git commit.

ACKNOWLEDGMENT

This material is based upon work supported in part by the National Science Foundation under Grant No. 2047120. This work was performed in part under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-POST-855166)

REFERENCES

[1] Miltiadis Allamanis. 2019. The Adverse Effects of Code Duplication in Machine Learning Models of Code. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Athens, Greece) *(Onward! 2019)*. Association for Computing Machinery, New York, NY, USA, 143–153. https://doi.org/10.1145/3359591.3359735

[2] Sid Black, Gao Leo, Phil Wang, Connor Leahy, and Stella Biderman. 2021. *GPT-Neo: Large Scale Autoregressive Language Modeling with Mesh-Tensorflow.* https://doi.org/10.5281/zenodo.5297715 If you use this software, please cite it using these metadata..

[3] Veselin A. Dobrev, Tzanio V. Kolev, and Robert N. Rieben. 2012. High-Order Curvilinear Finite Element Methods for Lagrangian Hydrodynamics. *SIAM Journal on Scientific Computing* 34, 5 (2012), B606–B641. https://doi.org/10.1137/120864672 arXiv:https://doi.org/10.1137/120864672

[4] AJ Kunen, TS Bailey, and PN Brown. 2015. KRIPKE-A massively parallel transport mini-app. *Lawrence Livermore National Laboratory (LLNL), Livermore, CA, Tech. Rep* (2015).

[5] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners. (2019).

[6] Frank F. Xu, Uri Alon, Graham Neubig, and Vincent J. Hellendoorn. 2022. *A Systematic Evaluation of Large Language Models of Code.* https://doi.org/10.5281/zenodo.6363556 https://arxiv.org/abs/2202.13169.