先进编译实验室
Advanced Compiler

《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

嘉宾：方泽贤

先进编译实验室
Advanced Compiler

论文信息

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
dnicho@umd.edu,marathe1@llnl.gov,harshitha@llnl.gov,tgamblin@llnl.gov,bhatele@cs.umd.edu

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 [3, 6, 8]. 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 [7], GPT-Neo [2], and PolyCoder [9], 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 ≈ 2GB of data with only ≈ 1.6GB being left afterwards.

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

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

- 作者信息

马里兰大学、劳伦斯利弗莫尔国家实验室

- 论文来源

2023年的SC会议

目录

01 背景与概述

02 数据收集与处理

03 微调方法

03 下游任务与评测指标

03 模型评测

背景与概述

代码大模型相关任务：

代码完成、恶意软件检测、代码重构、代码注释等

当前问题：

基于LLM的编码工具在革命性的软件开发中发挥了重要作用，但主流模型并没有针对高性能计算（HPC）问题进行设计、训练或测试。

1. 背景与概述

HPC-Coder：

只需为模型提供先前的上下文作为标记序列，然后让它生成新的标记，直到达到某个停止阈值。先前的上下文通常是自然语言注释，后跟函数声明，然后生成token，直到函数完成。

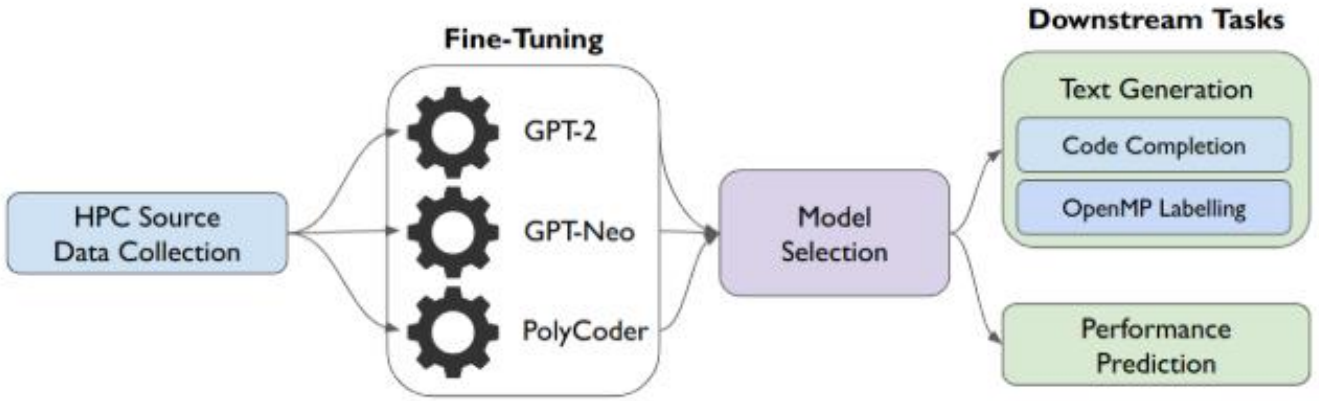

图1 训练HPC模型步骤概述

先进编译实验室
Advanced Compiler

2. 数据收集与处理

2.1 HPC源代码数据

数据集来源：Github代码仓库

数据提取限制：
（1）C/C++ 标记为主要语言
（2）stars数>=3
（3）HPC相关主题进行过滤
（4）收集c/c++扩展名源文件

图2 每种文件类型中代码行的分布

2.2 数据预处理

处理方法：
（1）根据内容的哈希值删除重复文件
（2）过滤大于 1 MB 的源文件
（3）过滤包含少于 15 个由语言词汇定义的标记的文件

Filter	# Files	LOC	Size (GB)
None	239,469	61,585,704	2.02
Deduplicate	198,958	53,043,265	1.74
Deduplicate + remove small/large files	196,140	50,017,351	1.62

表1 HPC源代码数据集属性

Miltiadis Allamanis. 2019. The Adverse Effects of Code Duplication in Machine Learning Models of Code. In Proceedings ofthe 2019ACMSIGPLANInternational 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

3. 微调方法

3.1 微调模型选择

选择要求：

（1）充分建模语言数据
（2）合理训练并部署下游任务
（3）模型和数据集开源
（4）~~codex~~等

Model	# Param	Layers	Hidden Size	Window Size	Pre-Training Set
GPT-2 [28]	1.5B	48	1600	1024	WebText [18]
GPT-Neo [9]	2.7B	32	2560	256	Pile [16]
PolyCoder [34]	2.7B	32	2560	2048	Source Code

表2 评估模型的描述

先进编译实验室
Advanced Compiler

3.2 微调方法

部分设置：

（1）使用 DeepSpeed 的Trainer 接口作为后端来优化训练；

（2）在具有 AMDEPYC 7763 CPU、512 GB 内存和四个 40 GB NVIDIA A100 GPU 的单个节点上对其进行微调；

（3）通过 ZeRO 内存优化，所有模型都完全适合单个 A100，因此使用纯数据并行性进行训练；

（4）AdamW优化器用于更新模型权重并最小化损失。学习率：5e-5，adam参数β1=0.9，β2=0.999；

（5）每1000个优化器步骤，我们还会在验证数据集上运行模型，并记录预测token时的困惑和准确性；

（6）验证数据集占完整数据集的 5%。

4. 下游推理任务和评估指标

4.1 代码完成

Name	Description	Seq.	OpenMP	MPI
Average	Average of an array of doubles	✓	✓	✓
Reduce	Reduce by generic function foo	✓	✓	✓
Saxpy	Saxpy	✓	✓	✓
Daxpy	Daxpy	✓	✓	✓
Matmul	Double precision matrix multiply	✓	✓	✓
Simple Send	Send MPI message			✓
Simple Receive	Receive MPI message			✓
FFT	Double precision FFT	✓	✓	✓
Cholesky	Single precision Cholesky factorization	✓	✓	✓
Ping-pong	MPI ping-pong			✓
Ring pass	MPI ring pass			✓

表3：代码生成测试。Sequential、OpenMP和MPI表示测试是否包括具有该并行后端的版本。

Prompt:

```
/*
 multiply scalar float a by vector x and add to y
 vectors x and y are length N
 use OpenMP to compute in parallel
*/
void saxpy(float *x, float *y, float a, int N) {
```

Output:

```
        #pragma omp parallel for
        for (int i = 0; i < N; i++) {
                y[i] += a * x[i];
        }
}
```

图3 sappy的共享内存并行实现的示例提示和输出

评估指标：HumanEval改编版

4. 下游推理任务和评估指标

4.1 代码完成

$$\text{pass@}k = 1 - \binom{N_p - c_p}{k} \Big/ \binom{N_p}{k} \quad (1)$$

$$\text{average pass@}k = \frac{1}{P}\sum_{i=1}^{P}\left[1 - \frac{\binom{N_i - c_i}{k}}{\binom{N_i}{k}}\right] \quad (2)$$

为了编译生成的代码，使用带有“-O2 -std=c++17 -fopenmp”标志的 g++。对于需要 MPI 的测试，我们使用 OpenMPI mpicxx 编译器。对于需要 OpenMP 或 MPI 的测试，我们仅在使用相应的并行框架来计算结果时才将其表示为正确。

4.2 预测 OpenMP 编译指示

数据集：

使用 HPC 代码数据集中的 OpenMP 编译指示创建每个 for 循环的数据集。还包括 for 循环之前的 500 个上下文标记。这会产生一个包含 13,900 个样本的数据集。

设置：

每个模型都在这个较小的数据集上训练 3 个时期（遍历整个数据集）。为了防止过度拟合，我们使用 3×10^{-5} 的起始学习率。在训练期间，留出 10% 的数据集用于验证。

评估指标：

（1）句法。为了衡量语法的正确性，我们将生成的编译指示与实际的编译指示进行文本等效性比较

（2）功能。由于我们无法从数据集中运行任意循环，因此我们通过将生成的编译指示与实际的编译指示进行比较来测量功能的正确性，同时忽略对功能没有贡献的差异。

4. 下游推理任务和评估指标

4.3 相对性能预测

数据集：

Kripke和 **Laghos**应用程序的每次提交。尽最大努力自动构建和运行每个提交，并收集总共 830 个提交的性能结果。然后，我们使用 90% 的数据训练模型，另外 10% 的数据留作评估。

评估指标：

使用模型来预测版本控制历史中的性能下降。为了实现这一点，我们为 Git 提交之前和之后的代码区域提供了模型文本。这些代码通过分隔它们的唯一标记连接起来，即 <COMMIT>。模型评估指标为慢（负）或快（正）。

5. 模型评测

5.1 代码完成

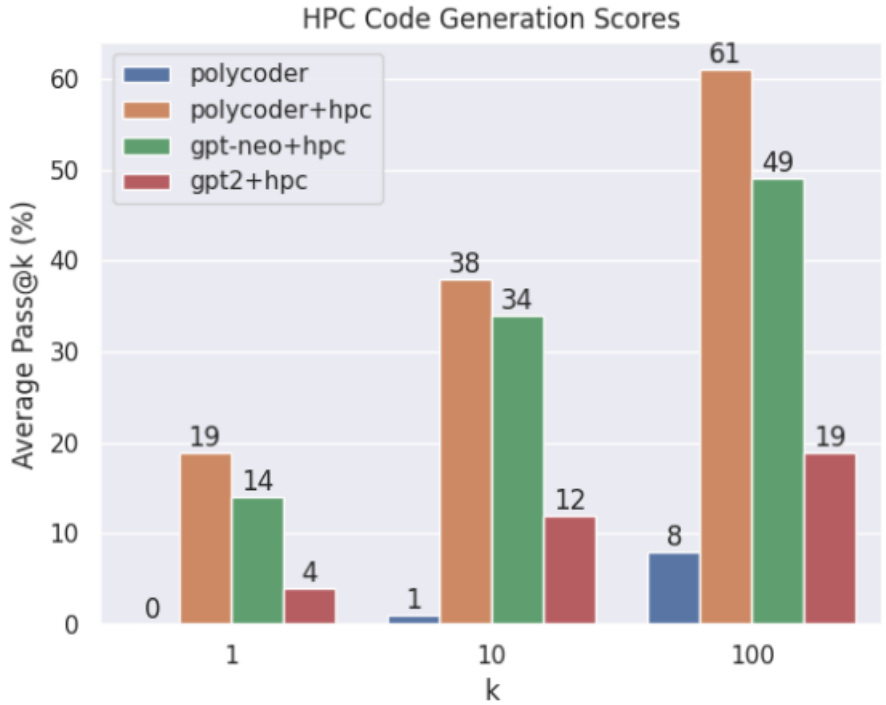

图4 HPC 特定功能的代码生成模型比较。这些簇代表 k = 1、10 和 100 的平均 pass@k 分数。百分比越高越好

Prompt:

```
/*
 Compute the sum of the array X and return the sum.
 X has N elements.
 Use OpenMP to compute the sum in parallel.
*/
float sum(float *X, int N) {
```

PolyCoder:

```
    float sum = 0.0f;
    for (int i = 0; i < N; i++)
        sum += X[i];
    return sum;
}
```

PolyCoder+HPC:

```
    float sum = 0.0f;
    #pragma omp parallel for reduction(+:sum)
    for (int i = 0; i < N; i++)
        sum += X[i];
    return sum;
}
```

图5 PolyCoder（中）和 PolyCoder+HPC（下）的 OpenMP 输出示例

5. 模型评测

5.2 预测 OpenMP 编译指示

两个模型在这两种指标都达到了不错的效果，且见过大量 OpenMP 代码的模型在生成编译指示方面做得更好。

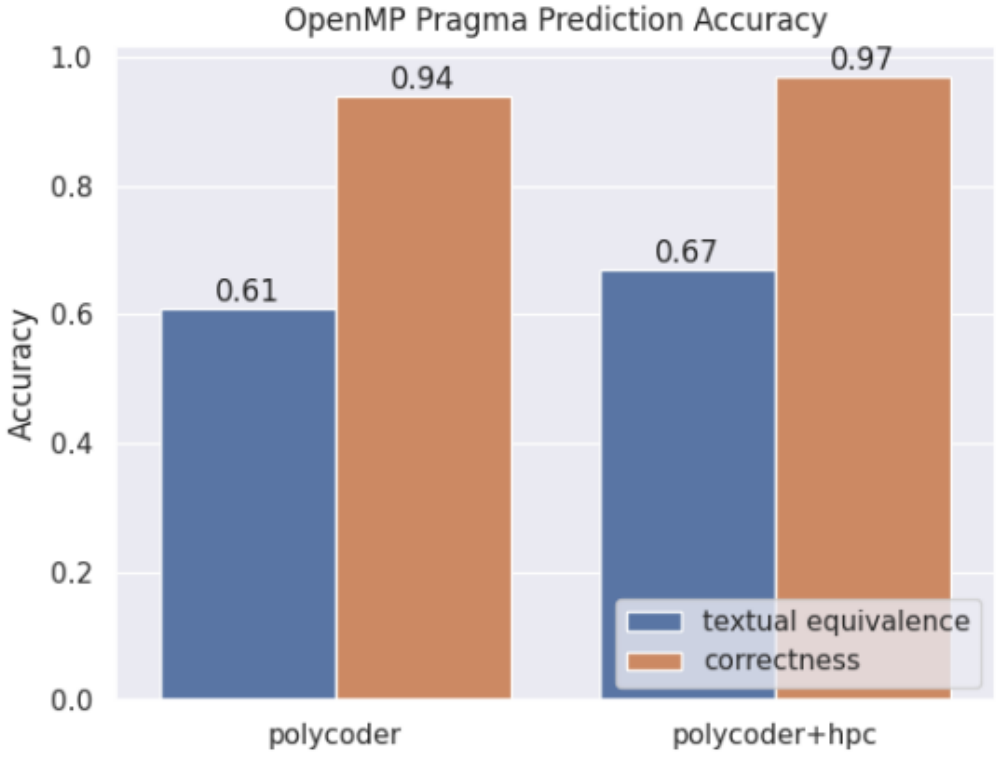

图6 预测OpenMP编译指示的模型比较

5.3 相对性能预测

这些模型能够将其先前的语言理解与代码的性能相关属性关联起来。这意味着我们可以利用 LLM 和微调来建模代码性能，而无需收集大量数据。

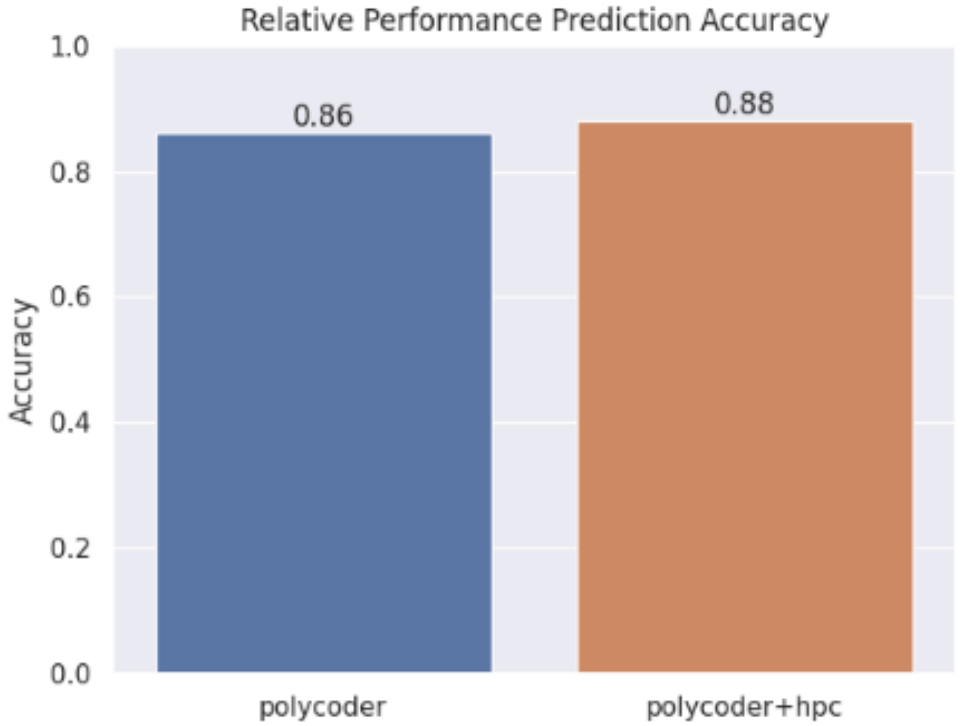

图7 预测代码更改相对性能的模型比较

AdvancedCompiler

Tel: 13839830713